UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de Cliente web integrado de gestión de contenidos y de flujos de trabajo sobre los motores de Pixelware Solutions y jBPM

Alberto Pérez Navarro
Tutor: Safwan Nassri
Ponente: Álvaro Ortigosa

Mayo 2016

Desarrollo de Cliente web integrado de gestión de contenidos y de flujos de trabajo sobre los motores de Pixelware Solutions y jBPM

AUTOR: Alberto Pérez Navarro
TUTOR: Safwan Nassri
PONENTE: Álvaro Ortigosa

Empresa Pixelware S.A
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2016

Resumen

Este Trabajo de Fin de Grado se centra en la participación en un proyecto real de empresa a gran escala, en el que se ha realizado una remodelación, migración de lenguaje y unificación de varios software obsoletos.

La finalidad principal es mejorar tanto la interfaz anticuada del lado Cliente, como la eficiencia del lado Servidor; para ello se han utilizado las últimas tecnologías programación web como *AngularJS, Bootstrap* y *Typescript* siguiendo la arquitectura *Single Page Application (SPA)*. En cuanto al Servidor, se ha desarrollado siguiendo la arquitectura *RESTful WebApi*.

La nueva aplicación web ofrece al usuario una interfaz mejorada totalmente *responsive* y una navegación más ligera debido a la reducción de carga del Servidor.

Para tener una visión global y tocar todas las áreas, en este Trabajo de Fin de Grado se han realizado una serie de módulos del proyecto general llevando a cabo todas las fases del ciclo de vida.

Cada módulo comprende unas fases previas de investigación del software obsoleto, reuniones con desarrolladores de la empresa y posteriormente el desarrollo del Cliente y el Servidor.

Se ha usado como gestor de contenidos el *Motor ECM* de la empresa *Pixelware S.A*, que mediante las librerías *PixelwareApi* interactúa con la base de datos y *Content Server* para la descarga de documentos. Adicionalmente también se han añadido algunas funcionalidades al *Motor ECM*.

Finalmente se han realizado unas pruebas concluyentes comparando ambos software para corroborar las motivaciones iniciales.

Palabras clave

Remodelación, migración de lenguaje, unificación, Software Obsoleto, AngularJS, Bootstrap,Typescript, Single Page Application (SPA), RESTful WebApi, responsive, Motor ECM, Pixelware, PixelwareApi, Content Server.

Abstract

This bachelor thesis focuses on participation in a real project of large-scale enterprise, which has made a remodeling, language migration and unification of several obsolete software.

The main purpose is to improve both the outdated client-side interface, such as server-side efficiency. It has been used the latest web programming technologies like AngularJS, Bootstrap and Typescript following single-page application architecture (SPA). As for the server, it has been developed following the WebAPI RESTful architecture.

The new web application makes available a fully responsive user interface and improved navigation faster due to reduced server load.

To have a global vision and work in all areas, this thesis includes the creation of several modules of the overall project performing all phases of the life cycle.

Each module comprises the preliminary stages of research obsolete software, developers meetings with the company and later the development of the client and the server.

This project has used the ECM Motor of Pixelware Company as content management, which through software Library PixelwareApi interacts with the database and Content Server for downloading documents.Additionally, it has been necessary to add functionality to ECM Motor.

Finally, there have been some representative tests comparing both software to confirm the initial motivations.

Keywords

Remodeling, language migration, unification, obsolete software, AngularJS, Bootstrap,Typescript, Single Page Application (SPA), RESTful WebApi, responsive, ECM Motor, Pixelware, PixelwareApi, Content Server.

Agradecimientos

En primer lugar agradecer a mi tutor Safwan Nassri por darme la oportunidad de realizar este Trabajo de Fin de Grado en la empresa Pixelware S.A.

También agradecer a los desarrolladores de la empresa, en concreto Alberto Chacón y Andrés Ramos, que siempre han estado ahí para aconsejarme.

Por último y más importante, a mi familia que me han apoyado en todo momento y en concreto a mi madre que sin tener conocimientos de informática, me ha ayudado en todo lo posible.

Alberto Pérez Navarro

Mayo 2016

INDICE DE CONTENIDOS

1 Introducción 1
1.1 Motivación 1
1.2 Objetivos 2
1.3 Organización de la memoria 3
2 Estado del arte 5
2.1 Proyecto General Pixelware SolutionsSpa 5
2.1.1 Software obsoleto 5
2.1.2 Unificación y remodelación 5
2.1.3 Punto de partida y Módulos a realizar asociados al TFG 6
2.2 Tecnologías y lenguajes usados 7
2.2.1 AngularJS 7
2.2.2 TypeScript 8
2.2.3 Bootstrap 9
2.2.4 RESTful WebApi 9
2.2.5 Motor ECM Pixelware 10
3 Análisis 11
3.1 Requisitos Funcionales 11
3.1.1 Módulo Administración 11
3.1.2 Módulo Explorar 13
3.2 Requisitos No funcionales 14
4 Diseño 17
4.1 Arquitectura 17
4.1.1 Arquitectura global de la Aplicación 17
4.1.1.1 Cliente 17
4.1.1.2 Servidor RESTful 18
4.1.1.3 Motor ECM 18
4.1.2 Extensión del diseño 19
4.2 Modelo de datos 19
4.2.1 Clases 19
4.2.2 Base de datos 20
4.3 Diseño de la interfaz 21
4.3.1 Módulo Administración 21
4.3.1.1 Tablas Auxiliares 21
4.3.2 Módulo Explorar 22
5 Desarrollo 25
5.1 Modelo de datos 25
5.1.1 Motor ECM 25
5.1.2 Cliente-Servidor 25
5.1.2.1 Tablas Auxiliares 25
5.1.2.2 Listado de Registros 26
5.1.2.3 Búsquedas 27
5.2 Servidor 28
5.2.1 Motor ECM 28
5.2.1.1 Módulo Administración 28
5.2.1.2 Módulo Tareas 29
5.2.2 RESTful WebApi 29
5.2.2.1 Controllers 29

5.2.2.2 Providers 29
5.3 Cliente 34
5.3.1 Configuración y Rutas 34
5.3.2 Service 34
5.3.3 Template-Controller 35
5.3.3.1 Módulo Administración 35
5.3.3.2 Módulo Explorar 38
6 Integración y pruebas 43
6.1 Pruebas unitarias 43
6.2 Pruebas de integración 43
6.3 Pruebas de Regresión y Validación 43
6.4 Pruebas de sistema 43
6.4.1 Pruebas de rendimiento 44
7 Conclusiones y trabajo futuro 45
7.1 Conclusiones 45
7.2 Trabajo futuro 45
Referencias 47
Glosario 49
Anexos ..... - 1 -
A Script SQL ..... - 1 -
B Función recursiva simulación de búsquedas en árboles ..... - 2 -
C Summary Report Jmeter ..... - 3 -

INDICE DE FIGURAS

FIGURA 1: ESTRUCTURA DE MÓDULOS SOLUTIONSSPA 6

FIGURA 2: SINCRONIZACIÓN BIDIRECCIONAL *ANGULARJS* 7

FIGURA 3: BOTÓN HTML SIN ESTILOS 9

FIGURA 4: BOTÓN HTML CON ESTILOS BOOTSTRAP 9

FIGURA 5: DISEÑO DE CAPAS Y COMUNICACIÓN *SOLUTIONSSPA* 19

FIGURA 6: EQUIVALENCIA Y TRADUCCIÓN DE CLASES ENTRE CLIENTE Y SERVIDOR 20

FIGURA 7: DISEÑO INTERFAZ DE USUARIO TABLAS AUXILIARES DE MÓDULO ADMINISTRACIÓN . 21

FIGURA 8: DISEÑO INTERFAZ DE USUARIO MÓDULO EXPLORAR 23

FIGURA 9: MODELO DE DATOS PARA LA GESTIÓN DE TABLAS AUXILIARES 26

FIGURA 10: MODELO DE DATOS PARA LA GESTIÓN DEL LISTADO DE REGISTROS 27

FIGURA 11: MODELO DE DATOS PARA LA GESTIÓN DE BÚSQUEDAS DEL LISTADO DE REGISTROS .. 27

FIGURA 12: EXPANSIÓN DE NODOS DEL ÁRBOL DE VALORES JERÁRQUICOS 31

FIGURA 13: VISTA GESTIÓN DE TABLAS AUXILIARES, BÚSQUEDA DE VALORES JERÁRQUICOS EN EL ÁRBOL 37

FIGURA 14: VISTA DIÁLOGO CREACIÓN DE TABLA AUXILIAR 38

FIGURA 15: VISTA MENÚ DE SELECCIÓN DE TABLAS DE USUARIO 38

FIGURA 16: VISTA DIÁLOGO CONFIGURACIÓN DE VISUALIZACIÓN 39

FIGURA 17: VISTA BÚSQUEDA AVANZADA 41

FIGURA 18: VISTA LISTADO DE REGISTROS DEL MÓDULO EXPLORAR 42

INDICE DE TABLAS

TABLA 1: RESULTADOS DE LA PRUEBA DE RENDIMIENTO JMETER 44

1 Introducción

1.1 Motivación

Desde el nacimiento de la informática, los lenguajes de programación han ido avanzando y actualizándose continuamente hasta nuestros días creando así nuevas técnicas y lenguajes que facilitan el desarrollo y la optimización de software.
Los nuevos programas informáticos comienzan desarrollándose partiendo de dichas novedades pero, ¿qué pasa con todo el software obsoleto?, ¿dejamos de utilizarlo y creamos nuevos programas?, la respuesta no es unánime. Muchos desarrollos son muy útiles y es ahí donde se aplica la expresión "no reinventar la rueda", por ello en estos casos hay que realizar un proceso de actualización de dicho software con tecnologías modernas para no quedarse estancado en el sector.

Este proceso de "modernización" de un software conlleva a una remodelación total aunque la funcionalidad final sigua siendo la misma. En dichas modificaciones podemos encontrar cambios en la arquitectura, lenguaje de programación, comunicación entre elementos (Cliente-Servidor), pruebas y rendimiento final que hacen dos software completamente diferentes aunque, la funcionalidad sigue siendo la misma.

Muchas empresas del sector con bastante antigüedad poseen software muy valioso, ya que muchos usuarios dependen de ello y es por esto que se ven obligados a renovarlo como se comenta anteriormente en vez de desecharlo.

Cuando hablamos de modernización en el software no puede faltar el término "internet" y la importancia que tiene actualmente, ya que casi todo el software que se desarrolla tiene funcionalidades online que incluyen peticiones a algún Servidor. Por esto y debido a la multiplataforma, el uso de dispositivos móviles y los avances en la programación web, cada vez se desarrolla más software accesible desde los navegadores evitando instalar programas y acceder a los recursos de manera más sencilla.

Una vez comentados estos temas, es aquí donde surge la motivación de este Trabajo de Fin de Grado (TFG), donde se explica y argumenta la participación en un proyecto real de empresa en el que se unifican varios programas obsoletos en una sola aplicación web SPA. Dicha aplicación se desarrolla con las últimas tecnologías modernas en el desarrollo de aplicaciones web, que ofrecerán una experiencia de usuario mejorada así como un incremento del rendimiento y eficiencia final.

1.2 Objetivos

El objetivo de este TFG es dar el salto del desarrollo de prácticas de asignaturas en los laboratorios a saber afrontar un desarrollo real de manera profesional en un ámbito de empresa donde se trabaja en equipo; para que esto sea posible, es necesario aplicar todos los conceptos aprendidos a lo largo de la formación en el Grado de Ingeniería Informática así como el trabajo en las Prácticas Curriculares.

Principalmente se aplica las asignaturas referentes a la programación web y servidores (Sistemas informáticos I y II), bases de datos (Estructuras de datos), análisis y diseño (Análisis y Diseño de Software y Proyecto de Análisis y Diseño de Software) y pruebas (Ingeniería del Software) entre otras; por otro lado no es posible citar ninguna asignatura específica en concreto, ya que es el conjunto de ellas lo que ha proporcionado un conocimiento y habilidad para saber adaptarse a un problema/necesidad y saber resolverlo.

Como se trata de un proyecto en equipo entre desarrolladores de la empresa, ha sido necesario saber adaptarse e incorporarse analizando y estudiando lo diseñado y desarrollado hasta el momento fijando claramente cuál va a ser el punto de partida, la tarea y la distribución del trabajo a realizar.

El proceso de migración y unificación del que trata este trabajo consta de varias fases y objetivos. El objetivo más importante es mantener la funcionalidad original del software, para ello es imprescindible estudiar y analizar toda la funcionalidad entendiendo "que hace" para poder cambiar el "cómo lo hace".
Por otro lado se ha tenido que aprender el desarrollo de programación web con tecnologías modernas (*AngularJS, TypeScript, BootStrap*), la arquitectura *SPA* (*Single Page Application*), los servicios web *RESTful WebApi* y el *Motor ECM* de la empresa Pixelware (librería *PixelwareApi*, *Content Server* y base de datos) que se usa para la interacción con la base de datos. Es por esto por lo que se comenta anteriormente que no hay asignaturas específicas que se puedan aplicar a este trabajo, ya que dichas tecnologías y componentes no se estudian en la carrera y por ello es necesario un proceso de investigación para su posterior aplicación partiendo de la base de los conocimientos aprendidos en la carrera.

En un proyecto real como este, no es la misma persona la que realiza las fases de análisis, diseño, desarrollo, pruebas y mantenimiento. Pero dado que forma parte de este Trabajo de Fin de Grado, se han separado varios módulos del proyecto para que se pueda realizar cada una de las fases descritas partiendo de un diseño inicial que hay que saber aplicar y extender. Por ello otro de los objetivos es comprender el diseño inicial para poder ampliarlo correctamente siguiendo con la filosofía de la aplicación y poder desarrollar los módulos del proyecto asignados a este Trabajo de Fin de Grado. La toma de decisiones ha sido propia consultando al tutor en caso de duda.

Otro de los objetivos, pero no menos importante, es realizar pruebas concluyentes de la mejora de la experiencia del usuario final así como del incremento del rendimiento y eficiencia en comparación con el software obsoleto.

Finalmente, el objetivo general de lo anteriormente mencionado se resume a la participación en un desarrollo real de gran escala formando parte de un equipo de trabajo y asumiendo las responsabilidades que ello conlleva.

1.3 Organización de la memoria

Esta memoria está estructurada en secciones que describen todas las fases necesarias para entender el desarrollo de este TFG dentro del proyecto general. Cada sección está compuesta a su vez de subsecciones en las que se especifican con detalle los temas que se están tratando. Las secciones principales están compuestas por los siguientes puntos:

- **Estado del Arte:** Se comenta los programas obsoletos que se van a unificar y rediseñar así como el punto de partida y módulos asociados a este trabajo. Por otro lado se explican las nuevas tecnologías de programación web que se han usado y las mejoras que proporcionarán.

- **Análisis:** Dado que la funcionalidad del software es la misma, en esta sección se hace un repaso de las mismas que se van a mantener así como las nuevas características que se han añadido al proyecto referente a los módulos que forman parte de este TFG.

- **Diseño:** En esta sección se explica el diseño y estructura de módulos que se han llevado a cabo. También se comenta como está organizada toda la aplicación y cuales han sido los bloques desarrollados en este TFG sobre el proyecto. Y por último la descripción de interfaces de usuario con algunas maquetas.

- **Desarrollo:** Se explica cómo se ha procedido para el desarrollo de los módulos y las investigaciones que han sido necesarias para poder realizarlos.

- **Pruebas y Resultados:** Conjunto de pruebas y resultados para argumentar la necesidad previa al desarrollo y en qué aspectos ha mejorado.

- **Conclusiones y Trabajo Futuro:** Balance total del trabajo realizado y el futuro de la aplicación en un ambiente real.

2 Estado del arte

2.1 Proyecto General Pixelware SolutionsSpa

En esta sección se va a comentar cual es la finalidad y motivación del proyecto general en el que se ha participado así como los módulos desarrollados en este Trabajo de Fin de Grado.

2.1.1 Software obsoleto

La motivación del proyecto es unificar y remodelar software obsoleto de la empresa Pixelware, pero antes de analizar cómo se ha realizado, veamos de manera general de qué software se está hablando y cuáles son sus funcionalidades:

- Pixelware Solutions: Aplicación web escrita en ASP.NET que engloba y unifica las aplicaciones *Pixelware Search* y *Pixelware WorkFlow Client* en una única aplicación accesible desde el navegador. Está enfocada a usuarios.
    - Pixelware Search: Aplicación web escrita en ASP.NET y C++ encargada de mostrar y añadir registros asociados a fichas de la base de datos mediante formularios así como de sus relaciones. Además contiene una serie de funcionalidades complejas que permiten realizar todo tipo de filtros y búsquedas locales, globales y avanzadas ofreciendo la posibilidad de cargar y guardar dichos filtros. La información a la que se puede acceder depende de los permisos del usuario.
    - Pixelware WorkFlow Client: Aplicación web escrita en ASP.NET y C++ encargada de mostrar y crear flujos de trabajo asociados a un usuario e interaccionar con ellos. También ofrece un sistema complejo de búsquedas y filtros.
- Pixelware Control Manager: Aplicación de escritorio escrita en C++ enfocada a administradores para la gestión de la base de datos y comportamiento de la aplicación *Pixelware Search*, que ofrece a rasgos generales funcionalidades como crear fichas especificando sus campos y relaciones o crear usuarios especificando sus permisos. En cuanto a la administración de flujos de trabajo para *Pixelware WorkFlow Client* existe otra aplicación de escritorio llamada *Pixelware WorkFlow Manager* pero actualmente no se ha planificado su unificación con las anteriores ni su remodelación.

2.1.2 Unificación y remodelación

Una vez comentado el software obsoleto, se expone cuál es la idea de unificación de los mismos en una sola aplicación web llamada *Pixelware SolutionsSpa*.

A la aplicación antes mencionada *Pixelware Solutions* que englobaba las aplicaciones *Pixelware Search* y *Pixelware WorkFlow Client* se le añade la parte de administración que incluye la aplicación *Pixelware Control Manager*. Todo forma un nuevo bloque *Pixelware*

SolucionsSpa donde dichas aplicaciones ahora se conocen como como *Explorar* (referente a *Search*), *Tareas* (referente a *WorkFlow Client*) y *Administración* (referente a *Control Manager*).

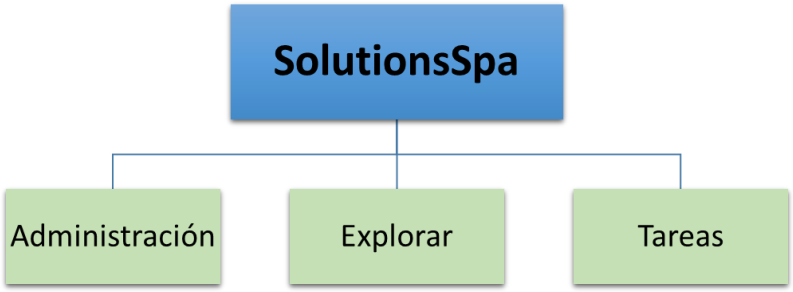


Figura 1: Estructura de módulos SolutionsSpa

De esta manera los usuarios o administradores pueden acceder al contenido de una forma más sencilla mediante el navegador simplificando sus necesidades.

Al igual que se han unificado las aplicaciones, también se han reescrito y rediseñado utilizando tecnologías modernas para incrementar la eficiencia y el rendimiento en conjunto. Se dejan de lado los lenguajes *ASP.NET* y *C++* para evolucionar a *AngularJS* en el Cliente y *C#* en el servicio con la arquitectura *RESTful WebApi*. Se ven en detalle estos lenguajes y su desarrollo en apartados posteriores.

2.1.3 Punto de partida y Módulos a realizar asociados al TFG

El proyecto tuvo unas fases iniciales de análisis y diseño en las que se fijó el esqueleto de la aplicación y a partir de ahí ha seguido un ciclo de vida incremental por módulos, refriéndose a estos como páginas o funcionalidades concretas independientes unas de otras. Tomando como origen el software obsoleto y teniendo en cuenta el esqueleto inicial, para cada módulo se realiza un ciclo de análisis, diseño y desarrollo que incrementa progresivamente la aplicación final.
Cada módulo o página incluye la funcionalidad completa de la misma, es decir, tanto la parte Cliente como la parte Servidor.

En cuanto al punto de partida para este Trabajo se dispone de varias funcionalidades desarrolladas referentes a los módulos *Administración* y *Tareas,* se pide el desarrollo de una página de la sección *Administración* así como de varias funcionalidades del módulo *Explorar.* Se explica en detalle en la sección de análisis.
La idea de este TFG es poder analizar, diseñar y desarrollar estas características siguiendo la filosofía inicial de la aplicación, siendo creativo y en caso de duda en la toma de decisiones, consultar al tutor.
Finalmente se realizan pruebas tanto unitarias como de integración para comprobar el correcto funcionamiento al fusionarlo con el resto del proyecto.

2.2 Tecnologías y lenguajes usados

En esta subsección se lleva a cabo un análisis y explicación de las tecnologías que se han utilizado para posteriormente entender mejor las secciones de diseño y desarrollo.
Principalmente se comentan los lenguajes modernos de desarrollo web que se han usado como principal punto de atracción, sus ventajas y un repaso general de su funcionamiento. Cada lenguaje o tecnología están separadas en subsubsecciones individuales pero formarían parte de dos bloques diferenciados: Cliente (*AngularJS, Typescript y Bootstrap*) y Servidor (*RESTful WebApi* y *Motor ECM Pixelware*).

2.2.1 AngularJS

Es un *framework JavaScript* para el desarrollo de aplicaciones web *front-end* creado por Google. Nació en 2009 pero empezó a ser popular a finales de 2012. Se basa en la filosofía *Single Page Application* (*SPA*) que utiliza una única página web cargada completamente sin necesidad de refrescarla, lo que resulta en una web más ligera y con menos llamadas al Servidor.
Anteriormente se usaba *JQuery* para la edición de la vista con funciones *JavaScript* que se añadían según la necesidad sin seguir ningún patrón, esto resultaba difícil de entender y mantener debido al desorden de código que desembocaba en el temido *SpaguettiCode*.

AngularJS nos ofrece un modelo de organización mucho más efectivo y fácil para desarrollar el código basado en el patrón *MVC* (*Model-View-Controller*) aunque ellos se definen como *MVW* (*Model-View-Whatever Works for you*). A esto se le añade la sincronización bidireccional (*Two-way Data Binding*) que permite sincronizar la vista y el modelo para que se puedan modificar los datos desde ambos sentidos.

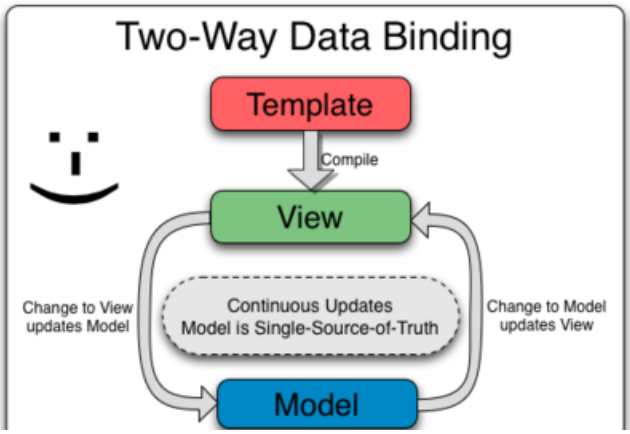


Figura 2: Sincronización bidireccional *AngularJS*

¿Cómo es posible este mecanismo?, extendiendo la sintaxis *HTML* de la vista mediante las nuevas directivas proporcionadas por el *framework*. *AngularJS* contiene decenas de directivas que nos proporcionan infinidad de ventajas, veamos un sencillo ejemplo usando la directiva *ng-model* para entender el patrón *MVC*:

- Modelo:
```
$scope.textoIntroducido
```
- Vista:
```
<div ng-controller='controladorTest'>
    <input type='text' ng-model='textoIntroducido'/>
    <p>Estás escribiendo lo siguiente: {{textoIntroducido}}</p>
</div>
```

- Controlador:

```
miAplicacion.controller(”controladorTest”, function($scope){
  $scope.textoIntroducido = ”Texto inicial”;

});
```

En el modelo se puede ver el atributo “textoIntroducido” contenido en el contexto *$scope*. Este contexto es un contenedor de atributos que usan las directivas de *AngularJS*, en él se guardan los modelos que se representan en la vista y también los atributos que se utilizan para manejar la lógica del controlador.
Se aprecia en el ejemplo que el modelo se puede modificar desde la vista y desde el controlador. En la vista (*HTML*) los valores introducidos en el “input” se almacenan en el modelo mediante la directiva *ng-model* y se representan mediante el doble corchete {{}}. En el controlador (JavaScript) inicialmente se puede fijar un valor del modelo o desarrollar funciones que modifiquen dicho valor al llamarlas y automáticamente se actualizaría la información en la vista.

Además de las directivas proporcionadas, podremos crear las nuestras desarrollando su funcionalidad deseada y usarlas en la vista.
Otros módulos que podemos usar en *AngularJS* son los *Service* que se encargan de comunicarse con el Servidor para enviar y obtener información que después será tratada por los controladores. Al igual que las directivas, tendremos los propios del *framework* y la posibilidad de crear nuevos *Service*. Veremos más funcionalidades en la sección de desarrollo.

En resumen, *AngularJS* nos permite separar el desarrollo de Cliente del de Servidor permitiendo el trabajo en paralelo, tiene gran modularidad pudiendo desarrollar bloques y funcionalidades independientes e inyectando sus dependencias en los controladores para la reutilización de código. La filosofía que sigue trata de realizar en Cliente parte del trabajo del Servidor liberando la carga del mismo y resultando de ello aplicaciones más ligeras que ofrecen un mayor rendimiento cuando el número de peticiones al Servidor es elevado.
Por otro lado el Cliente necesita procesar todo el código *JavaScript* y por ello es conveniente usar equipos más potentes.

2.2.2 TypeScript

Es un lenguaje de programación desarrollado por Microsoft que extiende la sintaxis de JavaScript. Se podría definir como un superconjunto de JavaScript que añade tipado estático y objetos basados en clases.
Nos proporciona el uso de herramientas que tenemos en lenguajes como C# y Java (clases, interfaces, tipos de datos, etc) sobre lenguaje JavaScript para facilitar la organización y desarrollo de proyectos grandes y ofrece un compilador que traduce el código a JavaScript informándonos de los errores.

La unión de AngularJS y TypeScript es una mezcla interesante que nos permite desarrollar un proyecto de gran escala, de lo contrario la organización y búsqueda de errores no sería tarea fácil y fuente de quebraderos de cabeza.

2.2.3 Bootstrap

Es un framework CSS desarrollado por Twitter para el diseño de aplicaciones web que permite aplicar estilos mejorando la experiencia del usuario, contiene librerías CSS que incluyen plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos basado en HTML y CSS.
Ofrece un mecanismo GRID basado en columnas que permiten realizar páginas web totalmente "responsive" multiplataforma, es decir, se adaptan al dispositivo y pantalla que las ejecute mostrando interfaces de usuario limpias. Además es compatible con la mayoría de navegadores (incluido IE).
La forma de usarlo es fijar las clases a elementos de la vista HTML así como tags o directivas para determinadas apariencias de diseño deseadas.
Veamos un sencillo ejemplo para un botón con indicador de número:

HTML

```
<button type="button">Usuarios
        <span>7</span>
</button>
```

Figura 3: Botón HTML sin estilos

HTML + Bootstrap

```
<button type="button" class="btn btn-primary">Usuarios
        <span class="badge">7</span>
</button>
```

Figura 4: Botón HTML con estilos Bootstrap

2.2.4 RESTful WebApi

Es un estilo de arquitectura para el diseño de servicio web y la interfaz entre Cliente y Servidor para obtener datos o indicar las operaciones sobre ellos en cualquier formato (XML, JSON, etc) sin las restricciones adicionales de los protocolos basados en patrones de intercambio de mensajes.
Se basa en una serie de puntos clave:

- Protocolo Cliente/Servidor sin estado: cada mensaje HTTP es independiente y suficiente para realizar la petición, es decir, no es necesario que el Servidor o Cliente recuerden el estado de la comunicación.
- La comunicación se realiza mediante un pequeño conjunto de operaciones básicas bien definidas, entre ellas POST, GET, PUT o DELETE con las que solicitaremos o enviaremos datos.
- La identificación de recursos (funciones del Servidor) se realiza con una sintaxis universal que se direcciona a través de su URI.

Con esta serie de funcionalidades se desarrolla el punto intermedio Servidor entre el Cliente y los motores ECM que acceden a la base de datos. Este punto intermedio escrito

en C# sigue la arquitectura RESTful estableciendo URI a cada recurso organizados en clases para poder acceder a ellos desde Cliente (AngularJS) mediante peticiones HTTP.
Para el envío o recepción de datos (objetos) y su representación tipada en clases se ha utilizado JSON en la transferencia desde Cliente y etiquetas de traducción en el Servidor que nos permiten obtener dichos datos en ambos sentidos sin perder la consistencia de clases. Todos estos mecanismos los veremos más en detalle en la sección de desarrollo.

2.2.5 Motor ECM Pixelware

La Gestión de Contenido Empresarial (*Enterprise Content Management, ECM* en inglés) es una metodología de organización y almacenamiento de documentos u otros contenidos ya sean en formato digital o impreso que se usa en un ámbito de organización empresarial.

En este trabajo se acceden a las librerías Pixelware.Api que forman parte de los motores ECM de la empresa Pixelware para satisfacer las necesidades de consulta y manipulación de datos desde el Servidor RESTful desarrollado.
Por otro lado también ha sido necesario el desarrollo de algunas funcionalidades en dichas librerías debido a la incorporación de nuevos requisitos en el proyecto como veremos más adelante.

Las funcionalidades que ofrecen las librerías Pixelware.Api son muy extensas debido a su uso en múltiples software de la empresa. Por esta razón y debido a que solo se ha accedido a ellas (además de la pequeña funcionalidad que se ha tenido que añadir) no se va a hacer hincapié en explicarlas en detalle y por tanto solo se comentan algunas de sus funcionalidades referentes a lo necesario en este trabajo.

Lo mismo ocurre con el diseño de la base de datos la cual organiza la información en registros pertenecientes a fichas con una serie de identificadores y relaciones determinados. Por otro lado también contiene información sobre los usuarios así como sus permisos y preferencias.

En resumen para este trabajo, los motores ECM ofrecen a simples rasgos la obtención de registros, su creación, edición y manipulación así como la información asociada a usuarios que en conjunto permitirá satisfacer las necesidades requeridas por el Servidor desarrollado.

3 Análisis

En esta sección se expone el análisis que se ha realizado para poder llevar a cabo la remodelación del software obsoleto así como para añadir las nuevas funcionalidades de este, como se ha comentado en secciones anteriores solo se muestran las funcionalidades referentes a este Trabajo y no de todo el proyecto.

En un principio se ha realizado una fase de investigación del software obsoleto interactuando con él; en esta fase se ha recogido los requisitos referente a las tareas que se van a llevar a cabo en este trabajo que incluyen parte de la remodelación de *Pixelware Search* y *Pixelware Control Manager* que ahora serán conocidas como *Explorar* y *Administración.*
A continuación se ha llevado a cabo una segunda fase en la que se han realizado entrevistas con otros desarrolladores del proyecto para confirmar y corregir los requisitos recogidos en la primera fase para comentar los nuevos requisitos que se han añadido o modificado en la nueva aplicación.
Por un lado tenemos los requisitos funcionales (RF) separados en dos subsistemas (Módulo Administración y Módulo Explorar) y por otro los requisitos no funcionales (RNF) que se agrupan en común.

3.1 Requisitos Funcionales

Los requisitos funcionales establecen el conjunto de comportamientos o funciones del sistema.

3.1.1 Módulo Administración

En este módulo de la aplicación destinado a Administradores se ha solicitado la creación de una página para administrar las tablas auxiliares que se usarán como novedad en los Motores ECM de Pixelware. La finalidad de dichas tablas es agrupar las *Tablas Asociadas* y *Tablas Jerárquicas* que usaban los motores ECM de Pixelware anteriormente. Una *Tabla Asociada* contiene una serie de valores con propiedades determinadas y una *Tabla Jerárquica* contiene una serie de valores con propiedades determinadas organizados en forma de árbol.
Estas tablas se usan para fijar valores de los campos de las tablas que visualizan los usuarios. Un ejemplo sería el siguiente, una tabla de usuario "Tabla_Usuario" contiene un campo "Localización". Para fijar dicho campo se hace uso (o un administrador la crea si es necesario) una tabla auxiliar jerárquica "Ciudades" organizadas en árbol de la forma Continente->País->Ciudad. Así cuando un usuario quiera añadir un registro a la tabla "Tabla_Usuario", a la hora de rellenar el campo "Localización", hará uso de la tabla "Ciudades" pudiendo seleccionar en ella un valor concreto.

[ADMINISTRACIÓN_RF1] Mostrar listado de tablas auxiliares
Se visualizará un listado de todas las tablas auxiliares existentes en la base de datos. Dado que dicha tabla es nueva, habrá que crearla en la base de datos y añadir en ella todas las Tablas Asociadas y Tablas Jerárquicas.

[ADMINISTRACIÓN_RF2] Búsqueda de tablas auxiliares
Se podrá buscar tablas auxiliares por título.

[ADMINISTRACIÓN_RF3] Creación de tablas auxiliares
Se podrá crear tablas auxiliares fijando su tipo (Asociada o Jerárquica) y sus propiedades (Referencia, Título y Descripción).

[ADMINISTRACIÓN_RF4] Edición de tablas auxiliares
Se podrá editar el título o descripción de una tabla auxiliar.

[ADMINISTRACIÓN_RF5] Eliminación de tablas auxiliares
Se podrá eliminar una tabla auxiliar, eliminando todos sus valores (asociados o jerárquicos según el tipo de la tabla) que contiene al realizar dicha acción.

[ADMINISTRACIÓN_RF6] Mostrar valores de una tabla auxiliar de tipo asociada de forma paginada
Se visualizarán todos los valores de una tabla auxiliar de tipo asociada al seleccionarla. Dichos valores se mostrarán de forma paginada con un límite de valores por página.

[ADMINISTRACIÓN_RF7] Buscar valores de una tabla auxiliar de tipo asociada
Se podrá buscar valores de una tabla auxiliar de tipo asociada trayendo los resultados paginados con un límite de valores por página.

[ADMINISTRACIÓN_RF8] Crear valores de una tabla auxiliar de tipo asociada
Se podrá crear valores de una tabla auxiliar de tipo asociada fijando sus propiedades (valor y visible).

[ADMINISTRACIÓN_RF9] Editar valores de una tabla auxiliar de tipo asociada
Se podrá editar valores de una tabla auxiliar de tipo asociada editando sus propiedades (valor y visible).

[ADMINISTRACIÓN_RF10] Eliminar valores de una tabla auxiliar de tipo asociada
Se podrá eliminar valores de una tabla auxiliar de tipo asociada.

[ADMINISTRACIÓN_RF11] Mostrar valores de una tabla auxiliar de tipo jerárquica en forma de árbol
Se visualizarán todos los valores de una tabla auxiliar de tipo jerárquica al seleccionarla. Dichos valores se mostrarán de forma de árbol pudiendo expandir o colapsar nodos.

[ADMINISTRACIÓN_RF12] Buscar valores de una tabla auxiliar de tipo jerárquica
Se podrá buscar valores de una tabla auxiliar de tipo jerárquica indicando el número de resultados de la búsqueda y mostrando las expansiones de nodos necesarias para llegar a cada resultado.

[ADMINISTRACIÓN_RF13] Crear valores de una tabla auxiliar de tipo jerárquica
Se podrá crear valores de una tabla auxiliar de tipo jerárquica fijando sus propiedades (valor y visible) y su posición en el árbol.

[ADMINISTRACIÓN_RF14] Editar valores de una tabla auxiliar de tipo jerárquica
Se podrá editar valores de una tabla auxiliar de tipo jerárquica editando sus propiedades (valor y visible).

[ADMINISTRACIÓN_RF15] Eliminar valores de una tabla auxiliar de tipo jerárquica
Se podrá eliminar valores de una tabla auxiliar de tipo jerárquica.

3.1.2 Módulo Explorar

La finalidad de este módulo destinado a Usuarios, es poder visualizar y crear registros de las tablas de usuario del sistema. Se pide un menú lateral para seleccionar o buscar una tabla y la visualización de los registros de la misma incluyendo una serie de complejos filtros de búsqueda.

[EXPLORAR_RF1] Mostrar un menú de tablas de usuario en forma de árbol
Se visualizará un menú lateral con todas las tablas de usuario en forma de árbol pudiendo expandir y contraer nodos.

[EXPLORAR_RF2] Buscar tablas de usuario en el menú
Se podrá buscar en el menú de tablas de usuario por título de la tabla indicando el número de resultados de la búsqueda y mostrando las expansiones de nodos necesarias para llegar a cada resultado.

[EXPLORAR_RF3] Mostrar registros de una tabla de usuario de forma paginada acorde a las preferencias de visualización del usuario (campos incluidos)
Se visualizarán los registros de la tabla seleccionada mostrando los campos incluidos acordes a las preferencias de visualización del usuario y con un límite de registros por página.

[EXPLORAR_RF4] Editar las preferencias de visualización de registros del usuario (campos incluidos y campos excluidos)
Se podrá editar las preferencias de visualización de registros fijando que campos se incluyen en la visualización (campos incluidos) y cuales no (campos excluidos).

[EXPLORAR_RF5] Restaurar las preferencias de visualización de registros del usuario (campos incluidos y campos excluidos)
Se podrá restaurar las preferencias de visualización de registros del usuario fijando los campos incluidos y campos excluidos por defecto de la tabla.

[EXPLORAR_RF6] Mostrar los campos excluidos de un registro
Se podrá visualizar los valores de los campos excluidos para un registro seleccionado.

[EXPLORAR_RF7] Búsqueda local de registros

Se podrán realizar búsquedas por campo/s indicando el texto a buscar y el filtro asociado (contiene, no contiene, empieza por, etc). Además se podrá fijar el texto a buscar seleccionando un elemento de una lista de valores actuales que incluirá los diferentes valores existentes para dicho campo en toda la tabla. Las búsquedas locales sólo serán de los campos incluidos en la visualización.

[EXPLORAR_RF8] Búsqueda global de registros
Se podrá realizar una búsqueda global de registros que buscará el texto fijado para todos los campos (incluidos y excluidos).

[EXPLORAR_RF9] Añadir criterio a la búsqueda avanzada de registros
Se podrá realizar una búsqueda avanzada de registros. Dicha búsqueda incluirá tantos criterios como se desee indicando su relación lógica entre ellos (AND u OR) y ofreciendo la posibilidad de agruparlos entre paréntesis. Cada criterio incluye el nombre del campo (incluido u excluido), el texto a buscar y su filtro asociado (contiene, no contiene, empieza por, etc). El texto a buscar variará según el tipo de campo que ofrecerá dinámicamente al usuario la forma de introducirlo.

- Tipo lista: Ofrecerá una lista seleccionable.
- Tipo fecha: Ofrecerá un calendario seleccionable.
- Tipo hora: Ofrecerá la hora seleccionable.
- Tipo asociado: Ofrecerá una ventana de selección múltiple de forma paginada permitiendo la búsqueda de valores.
- Tipo jerárquico: Ofrecerá una ventana de selección múltiple en forma de árbol permitiendo la búsqueda de valores.

[EXPLORAR_RF10] Editar criterio de la búsqueda avanzada de registros
Se podrá editar las características de un criterio de la búsqueda avanzada de registros.

[EXPLORAR_RF11] Eliminar criterio de la búsqueda avanzada de registros
Se podrá eliminar un criterio de la búsqueda avanzada de registros.

[EXPLORAR_RF12] Cambiar posición de criterios de la búsqueda avanzada de registros
Se podrá cambiar la posición de un criterio (subir o bajar) de la búsqueda avanzada de criterios.

3.2 Requisitos No funcionales

Los requisitos no funcionales establecen características del diseño e implementación del sistema.

[SOLUTIONSSPA_RNF1] Seguridad y Privacidad
Sólo se podrá acceder al contenido si el usuario está autenticado en la aplicación, por ello en cada subsistema o módulo se comprobará dicha propiedad antes de cargar el contenido.
Todos los datos introducidos por el usuario que interaccionen con la base de datos se realizarán con sentencias preparadas y en el caso de no poder realizar dichas sentencias se comprobará el texto siguiendo determinadas restricciones con expresiones regulares. De esta manera se evitará la inyección *SQL*.

[SOLUTIONSSPA_RNF1] Mantenibilidad
Se desarrollará cada módulo o capa de la aplicación de manera modular para poder reutilizar la funcionalidad mediante la inyección de dependencias de *AngularJS*. Aprovechando el uso de *TypeScript* se fomentará la creación de clases para mejorar la claridad del código, reusabilidad del mismo y compatibilidad con las clases del Servidor.

[SOLUTIONSSPA_RNF1] Portabilidad
La aplicación seguirá una filosofía de diseño *responsive* adaptable multiplataforma para poder ejecutarse correctamente en distintos equipos o dispositivos.

[SOLUTIONSSPA_RNF1] Volumen de datos
Para evitar la carga del Servidor se reducirán el número de llamadas al mismo y se minimizará el contenido transferido (paginación de listas, árboles colapsados, etc) y se hará uso de cachés.

[SOLUTIONSSPA_RNF1] Usabilidad
La aplicación tendrá una interfaz sencilla que proporcione una experiencia mejorada al usuario con respecto al software obsoleto. Se hará uso de *Tooltips* para ayudar al usuario y se seguirá el mismo estilo en todos los subsistemas para acelerar el aprendizaje de uso de toda la aplicación. Se minimizarán los errores guiando al usuario para realizar los procesos correctamente (advertencias de tamaño máximo de un campo, nombres de títulos existentes, etc) y mostrándole el resultado de las operaciones. Por último la aplicación tendrá una navegación sencilla que permitirá acceder a cualquier punto en menos de 3 clic.

4 Diseño

El diseño de un software es una de las fases más importantes del proyecto, que se realiza en función de las necesidades y comunicación entre agentes (Cliente-Servidor).
En esta sección se analizará en lo referente a la arquitectura, cual es el esqueleto de la aplicación que se tiene como punto de partida que nos ofrece el *framework* y cómo se ha extendido para poder realizar los módulos asociados a este trabajo. A continuación se expone en lo referente al modelo de datos, como se ha realizado la compatibilidad de clases para la comunicación Cliente-Servidor y las modificaciones realizadas en el *Motor ECM de Pixelware*. Por último se presentan maquetas de las diferentes pantallas con la que interaccionará el usuario.

4.1 Arquitectura

La arquitectura de un software indica cómo se separan y comunican los módulos de la misma y establece un esqueleto inicial para facilitar la modularidad y el desarrollo.

4.1.1 Arquitectura global de la Aplicación

La aplicación se ha desarrollado en *AngularJS* usando su patrón *MVC* que incluye las particularidades que se comentan en las primeras secciones de este documento, en el que se explica que el modelo puede ser modificado tanto por la vista como por el controlador. Por esta razón no se puede llevar una arquitectura de capas separando la capa de presentación de la lógica de negocio ya que son dependientes en el desarrollo. Se distingue entonces el diseño en lo referente a la parte Cliente en el que se separan los módulos que nos ofrece el *framework* y por otro lado el servicio *RESTful* que se encargará de comunicarse con el Cliente y acceder a los *Motores ECM de Pixelware* o a la *Caché*.

4.1.1.1 Cliente

Contiene toda la lógica y visualización que se ejecuta en el Cliente. Se usa un patrón específico separando los módulos en diferentes carpetas para facilitar el desarrollo a medida que se incrementa el tamaño del proyecto.

- **Templates:** Contiene los ficheros (*HTML)* con la parte visual (Vista) de la aplicación que representa el navegador. Aunque *AngularJS* funciona con una única página sin recargarla, se desarrolla cada parte de la aplicación en ficheros *HTML* independientes que se inyectarán y fusionarán dinámicamente en la página principal. Con esto se muestra al usuario las ventanas que solicita en cada momento de igual manera que ocurre con la navegación tradicional donde se redirigen enlaces. Los *Template* principales tienen asociado un *Controller*.

- **Controllers:** Contiene los controladores (*TypeScript*) que manejan la lógica de la aplicación en el lado Cliente, modifican e inicializan los datos y ofrecen funciones y comportamientos para los mismos. Los controladores interactuarán con la vista y viceversa y serán los encargados de comunicarse con los *Services*. Además

podremos inyectar dependencias de otros módulos para la reutilización de código. Al igual que los *Template* distinguiremos cada controlador según partes diferenciadas de la aplicación.

- **Services:** Contiene los servicios de Angular (*TypeScript*) de la aplicación y al igual que los *Template* y *Controller* distinguiremos cada servicio según las funcionalidades específicas que quiera desempeñar. Los servicios ofrecerán una serie de funciones para realizar las llamadas al Servidor *RESTful* que incluyen las peticiones *HTTP* que se solicitan y el tipo de datos devuelto. Serán llamados desde los *Controller*.

Además de estos grandes bloques tendremos clases para el modelo de datos, directivas, ficheros de traducción, ficheros de configuración y ficheros de enrutado.

4.1.1.2 Servidor RESTful

Contiene parte de la lógica que se ejecuta en el Servidor como punto intermedio entre el Cliente y el *Motor ECM de Pixelware*. Se usa un patrón específico separando los módulos en carpetas para facilitar el desarrollo a medida que se incrementa el tamaño del proyecto.

- **Controllers:** Contiene los controladores (*C#*) del Servidor en los que se incluyen los recursos del servicio *RESTful*. Cada función está identificada con una ruta independiente URI las cuales se solicitan desde el Cliente. Siguiendo la filosofía del punto anterior, se distingue cada fichero según las funcionalidades que se requieren del Servidor. En cada función se realizan unas comprobaciones de seguridad para identificar al usuario y se solicita o envía la información recibida del Cliente a los *Provider*.
- **Providers:** Contiene los proveedores (*C#*) del Servidor que serán encargados de interaccionar con el *Motor ECM de Pixelware* o la caché y devolver la información o resultado de la operación a los *Controller* (del Servidor), los cuales se comunicarán con el Cliente.

4.1.1.3 Motor ECM

Contiene parte de la lógica que se ejecuta en el Servidor y forma parte del último punto del camino en la comunicación entre Cliente y Servidor; está formado por una serie de librerías llamadas *PixelwareApi* que interactúan con la base de datos para obtener o manipular los datos y el Servidor de contenidos (*Content Server*) para importar o descargar documentos.

A continuación se muestra un esquema global de la comunicación entre capas y módulos de la aplicación.

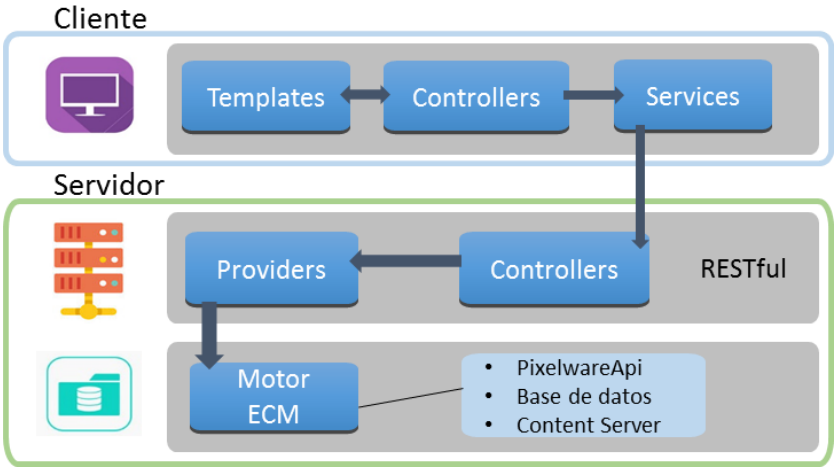


Figura 5: Diseño de capas y comunicación *SolutionsSpa*

4.1.2 Extensión del diseño

Una vez obtenidos los requisitos en la fase de análisis y con el esqueleto de punto de partida del framework, se crean los nuevos ficheros siguiendo la filosofía de diseño para poder desarrollar las nuevas funcionalidades. Como ya se comentó en secciones anteriores, el ciclo de vida es incremental y por tanto se sigue el proceso de análisis, diseño y desarrollo para cada bloque de funcionalidades determinada. En general se distinguen dos bloques principales, *Administración* y *Explorar*.

Las peticiones seguirán el "camino" *Template/Controller* (Cliente) => *Service* (Cliente) => *Controller* (Servidor) => *Provider* (Servidor) => *Caché/Motor ECM* representado en el esquema del punto anterior y su correspondiente vuelta de la petición. Por ello será necesario crear nuevos ficheros respetando el diseño para cada módulo/carpeta tanto de la parte Cliente como de la parte Servidor.

Se añaden *Templates* para las vistas, *Controllers (Cliente)* para el manejo de los datos, *Services (Cliente)* para la comunicación con el Servidor *RESTful*, *Controllers* (Servidor) para comprobaciones de seguridad e identificación de funciones con *URI*, *Providers (Servidor)* para la manipulación de datos, interacción con la caché y *Motor ECM* y por último se añaden algunas funcionalidades al *Motor ECM* para las nuevas *Tablas Auxiliares* como se explica en la sección de análisis.
Aunque se distinguen dos bloques diferenciados (*Administración* y *Explorar*) ambos comparten algunas funcionalidades y por ello y gracias al diseño modular de Angular se reutiliza código inyectando las dependencias de algunos módulos y usando directivas como se explica en la sección de desarrollo.

4.2 Modelo de datos

4.2.1 Clases

Dada la existencia de lógica tanto en el lado Cliente como en el lado Servidor, para garantizar la correlación de los datos en ambos sentidos se usan clases equivalentes

TypeScript (*.ts*) y *C#* (*.cs*) que mediante *serialización* hacen posible mantener los datos tipados en clases facilitando la manipulación de los mismos.
Por ejemplo, en un punto dado de la aplicación en el Cliente donde se requiere enviar un conjunto de datos tipados en clases al Servidor, se realiza una petición *HTTP* con los datos serializados en *JSON*. En el Servidor *RESTful* se tiene las mismas clases definidas para los datos tipados identificando cada atributo con las librerías de *serialización;* de esta forma en la función que recibe la petición *HTTP REST* se dispone de los mismos datos originales en tipados en clases. En la respuesta a la petición de devuelve un objeto *JSON* que *TypeScript* sabe interpretar para obtener los datos en clases. El esquema de flujo de manera general se muestra a continuación.

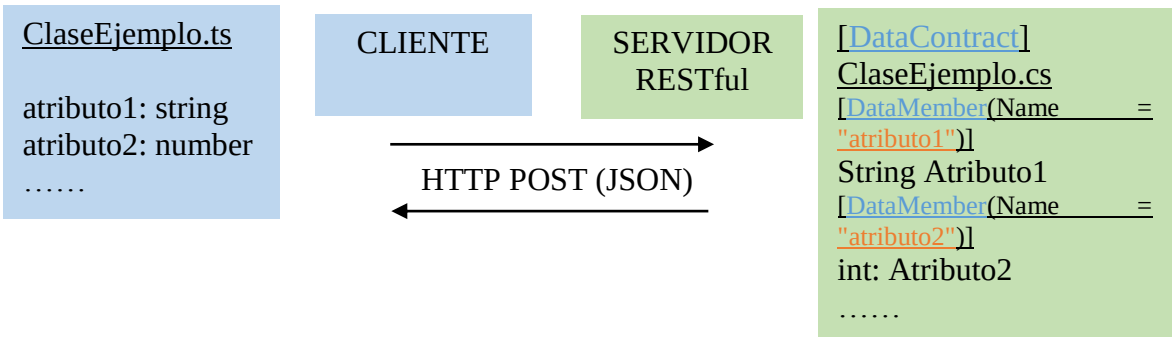


Figura 6: Equivalencia y traducción de clases entre Cliente y Servidor

4.2.2 Base de datos

En este trabajo al igual que en el proyecto global se utiliza el *Motor ECM de Pixelware* que ofrece librerías para obtener y manipular información de la base de datos, por esta razón no se detalla el diseño de la misma. Pero debido a la necesidad de creación de la nueva tabla *Tablas Auxiliares*, explicada en la sección de Análisis para agrupar a las *Tablas Asociadas* y *Tablas Jerárquicas*, ha sido necesario añadir funcionalidades al *Motor ECM*.

Se crea una nueva tabla *PWSAUXILIARES* que tendrá los siguientes atributos:

- **Referencia:** Clave primaria, nombre de la tabla (auxiliar o jerárquica) a la que hace referencia.
- **Título:** Título de la tabla a la que se hace referencia.
- **Descripción:** Descripción de la tabla a la que se hace referencia.
- **Tipo:** Tipo de la tabla a la que se hace referencia (auxiliar o jerárquica).

Cada tabla a la que se hace referencia (auxiliar o jerárquica) contiene una serie de valores con la siguiente estructura:

- **Auxiliares:** Comienzan por *PWAS*, por ejemplo *PWASCODIGOS* y tienen los siguientes atributos:
  - **Id:** Clave primaria, identificador del valor.
  - **Valor:** Valor.
  - **Mostrar:** Indica si se mostrará el valor o no.
- **Jerárquicas:** Comienzan por PWJS, por ejemplo *PWJSPROVINCIAS*, destinadas para representarse en forma de árbol. Tienen los siguientes atributos:

- **Id:** Clave primaria, identificador del valor.
- **Nodo:** Valor.
- **Parent:** Identificador del padre del valor, será 0 en caso de no tener padre.
- **Mostrar:** Indica la visualización del valor.

4.3 Diseño de la interfaz

El diseño de la interfaz es similar al existente en el software obsoleto, pero mejorando la experiencia del usuario gracias a las herramientas que nos proporciona *Bootstrap*. Es plenamente *responsive* por lo que se adapta a las distintas pantallas donde se ejecute la aplicación. Por otro lado se tienen que volver a desarrollar las vistas dado que en *AngularJS* se implementan funcionalidades que afectan al modelo de datos, por lo que las interfaces de usuario del software obsoleto solo sirven de referencia. Las páginas creadas muestran un estilo similar al resto de páginas para seguir en la línea del proyecto general. A continuación se explican las interfaces de usuario de cada módulo de la aplicación desarrollado en este TFG.

4.3.1 Módulo Administración

4.3.1.1 Tablas Auxiliares

Esta página proporciona al usuario las funcionalidades de gestión de *Tablas Auxiliares*. Contará con un listado de tablas permitiendo su búsqueda y selección, y además se dispone de un menú para crear, modificar o eliminar tablas.
Una vez seleccionada una tabla, se muestra a su derecha un listado con los valores que contiene. Si la tabla es de tipo Asociada, los valores se presentan en forma de lista paginada y si es de tipo Jerárquica se muestran en forma de árbol. En ambos tipos se pueden buscar valores y contienen un menú para crearlos, modificarlos o eliminarlos.

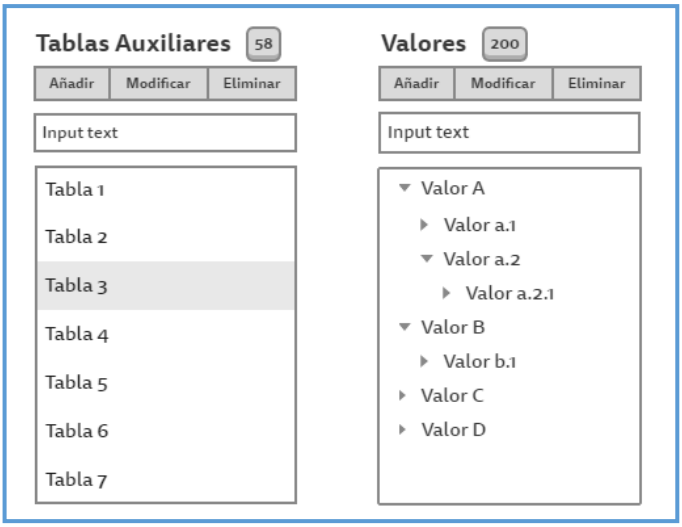


Figura 7: Diseño interfaz de usuario Tablas Auxiliares de Módulo Administración

En caso de realizar alguna acción de los menús (Añadir, Modificar o Eliminar) de una *Tabla Auxiliar* o de *Valores*, se muestra un diálogo con un formulario según el tipo de acción, que oculta parcialmente la pantalla principal. Dicho formulario informará al usuario de la acción que va a realizar y posteriormente se muestra el resultado de la misma. Se recuerda que la aplicación no recarga la página, por lo que toda modificación es reflejada automáticamente.

4.3.2 Módulo Explorar

Este módulo incluye una página principal de visualización de registros de *Tablas de Usuario* con varias herramientas de búsqueda (local, global y avanzada) para ayudar al usuario a encontrar registros. Previamente hay que seleccionar una tabla del árbol del menú lateral de la página, el cual también ofrece la posibilidad de búsqueda.

Una vez seleccionada se muestra sus registros en forma de tabla **paginada** visualizando solo los campos definidos en la **configuración de visualización** de dicha tabla para dicho usuario (o los campos por defecto si no hay ninguna configuración guardada), el usuario puede cambiar esta configuración en la propia página. El resto de **campos no visibles** se pueden visualizar para un registro cualquiera mediante una de las opciones que ofrece la interfaz.
Si un registro tiene **documentos** asociados, se da la opción al usuario de la descarga de los mismos.

La interfaz ofrece tres tipos de búsqueda:

- **Local:** Situada sobre cada cabecera de cada campo de la tabla, como se explica en la sección de análisis ofrece la búsqueda indicando el texto y filtro a aplicar sobre el campo. Además uno de los posibles filtros es la opción *Valores actuales* que permite fijar el texto seleccionando un valor de los distintos valores existentes para dicho campo y dicha tabla (con la apertura de un diálogo).

- **Global:** Situada en la parte superior de la página, realiza una búsqueda global sobre todos los registros y campos.

- **Avanzada:** Muestra un desplegable oculto que permite realizar una búsqueda fijando múltiples variables. Como se especifica en la sección de análisis, se añadirán con relaciones lógicas entre ellos (AND u OR) o paréntesis. Cada criterio incluye el nombre del campo, el tipo de filtro y una entrada de texto que es de forma distinta según el tipo de campo (texto, fecha, asociado, jerárquico, entero, etc). En casos como los de tipo fecha se proporciona un calendario y en casos más complejos como los de tipo asociado o jerárquico se muestra una opción para abrir un diálogo que permite la selección de valores (en forma paginada o en forma de árbol al igual que en la página de Tablas Auxiliares).

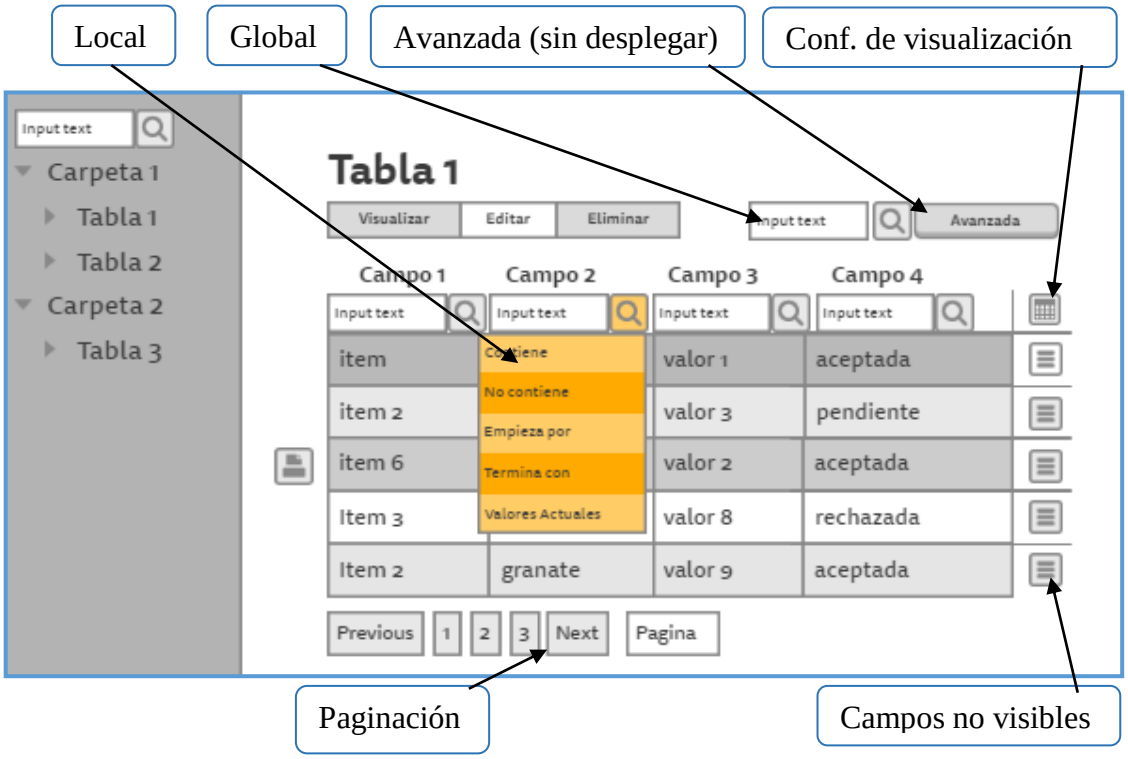


Figura 8: Diseño interfaz de usuario Módulo Explorar

Por otro lado, algunas opciones del menú superior de la página (Visualizar, Editar, Eliminar, Nuevo e Informes) ofrecen unas funcionalidades con formularios dinámicos creados en el Servidor que se están desarrollando actualmente fuera de este TFG, por lo que se ha realizado otra página secundaria a esta página principal que permite una integración total y sencilla con dichas funcionalidades como se explica en la sección *Pruebas de Integración.*

Cualquier acción ya sea un cambio en la visualización de campos o resultado de búsqueda se refleja sin recargar la página manteniendo la filosofía SPA. Por otro lado la apertura de diálogos comentados tanto en el módulo *Administración* como en el módulo *Explorar* ocultan parcialmente el contenido de la página principal enfocando toda la atención en ellos. Anteriormente en el software obsoleto, estos diálogos eran inexistentes, en el caso de requerirse se abría una ventana nueva o se mostraba la información en cuadros adicionales dentro de la página principal.

5 Desarrollo

En esta sección se explica cómo se ha llevado a cabo la implementación de los dos módulos de este TFG en el proyecto (*Tablas Auxiliares* del módulo *Administración* y módulo *Explorar*). Se expone el proceso de desarrollo de los bloques del Servidor y de los bloques de la parte Cliente, así como el modelo de datos que se ha usado para manejar la información. Por otro lado, debido al diseño en bloques que reutiliza código y funcionalidades, sólo se ha separado la explicación de los módulos *Administración* y *Explorar* en la parte Cliente.

5.1 Modelo de datos

A continuación se exponen las clases que se han desarrollado en este TFG así como las desarrolladas que se han usado para la implementación. Se incluyen en esta sección de desarrollo ya que aun teniendo el análisis y diseño claro de la aplicación, se han creado en la fase de desarrollo según las necesidades encontradas.

5.1.1 Motor ECM

Se crea en las librerías del *Motor ECM* la clase abstracta *AuxiliarTable* de la cual heredan las clases existentes *AssociatedTable* y *HierarchicalTable;* esta clase contiene los mismos atributos que se han incorporado a la base de datos con la tabla *PWSAUXILIARES* como se explica en la sección de Diseño. Contiene los siguientes atributos:

- `string _nameTable:` referencia de la tabla.
- `string _title:` título de la tabla.
- `string _description;` descripción de la tabla.
- `AuxiliarTableType _type;` tipo de la tabla (Asociada o Jerárquica).
- `SchemeNode _node:` atributo de control para interaccionar con el *Motor ECM*.
- `UserFileSystem _userFileSystem:` atributo de control para interaccionar con el *Motor ECM* referente a las propiedades de usuario.

5.1.2 Cliente-Servidor

Como se explica en secciones anteriores, se ha buscado en este Proyecto mantener la correlación de datos tipados entre diferentes capas (Cliente y Servidor), por ello las clases principales tienen en ambos sentidos los mismos atributos y se realiza la traducción mediante *serialización* y *JSON*. Estas clases solo contienen estructuras de datos y constructores ya que todas las funciones se realizan mediante las capas explicadas en la sección de diseño. A continuación se exponen las clases y sus relaciones más importantes.

5.1.2.1 Tablas Auxiliares

Se crea la clase *WebAuxiliarTable* tanto en Cliente (*Typescript*) como en Servidor (*C#*) para la información de las *Tablas Auxiliares,* dicha tabla tiene los atributos principales de la mencionada en el punto anterior del *Motor ECM* (*Auxiliar Table).*

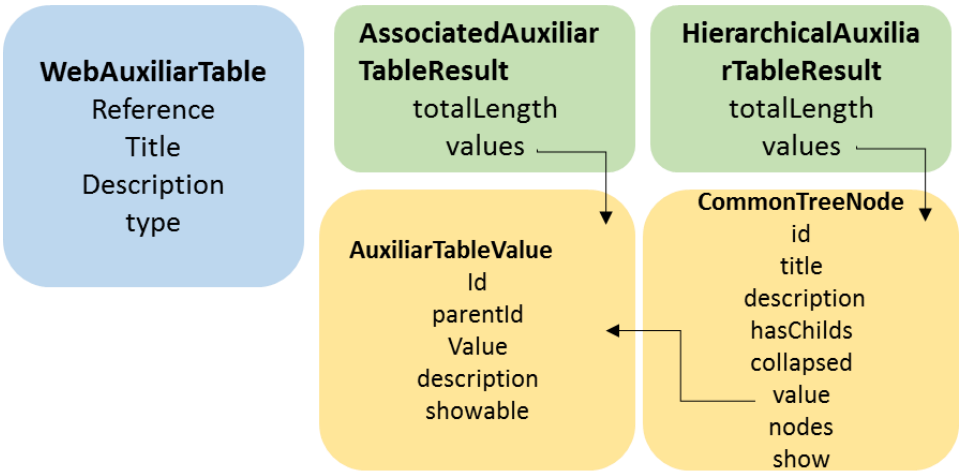


Figura 9: Modelo de datos para la gestión de Tablas Auxiliares

Para el manejo y representación de los valores de las *Tablas Auxiliares* se crean las clases *AssociatedAuxiliarTableResult* (valores asociados) e *HierarchicalAuxiliarTableResult* (valores jerárquicos). Estas tablas contienen un listado de valores de clase *AuxiliarTableValue* (valores asociados) o de clase *CommonTreeNode* (valores jerárquicos). La estructura de datos de las clases *AssociatesAuxiliarTableResult, HierarchicalAuxiliarTableResult, AuxiliarTableValue y CommonTreeNode* se ha dado ya desarrollada para este TFG para la parte Servidor y se ha desarrollado en la parte Cliente.
La estructura CommonTreeNode se usa para el manejo y representación de árboles. Su atributo *value* es de tipo objeto (o *any* en *Typescript*) y se usa para fijar si es necesario el valor AuxiliarTableValue con propiedades útiles como *parentId* para la búsqueda de padres en los árboles.

Los listados de valores (asociados o jerárquicos) contenidos en estas clases se obtienen de la *caché* (desarrollada fuera del TFG), que a su vez interactúa con el *Motor ECM* si el contenido no está *cacheado*.

5.1.2.2 *Listado de Registros*

Los registros de las tablas de usuario que se muestran en la página principal del *Módulo Explorar* se obtienen de las librerías del *Motor ECM*. Estas librerías nos permiten obtener los registros tras especificar varios atributos de control del *Motor ECM* en un listado de objetos de tipo *Record*. Este tipo de objeto es muy extenso y tiene atributos innecesarios para el resto de bloques de la aplicación (*RESTful WebApi* y Cliente) por lo que se han creado varias clases (en Cliente y Servidor) para manejarlos y representarlos.

Los registros como se detalla en la sección de Diseño, se muestran en forma de tabla paginada por lo que se crea una estructura siguiendo la filosofía de celdas de una tabla. La clase principal *FileRecord* contiene la información general de la *Tabla de Usuario* y el atributo *table* de tipo *FileRecordInstancesTable* que contiene el propio *grid* de celdas. Dicha clase *FileRecordInstancesTable* está formada por una lista de cabeceras (de tipo *TableHeaderCell*) y de filas (de tipo *TableBodyRow*) de la tabla. La clase *TableHeaderCell* tiene atributos para definir y especificar los campos de la tabla y un atributo auxiliar *auxiliarField* de tipo objeto (o any en *Typescrip*) que se explica más adelante. La clase *TableBodyRow* tiene un idenficador, una lista de documentos (*WebDocument*) y una lista de celdas (de tipo *TableBodyCell*).

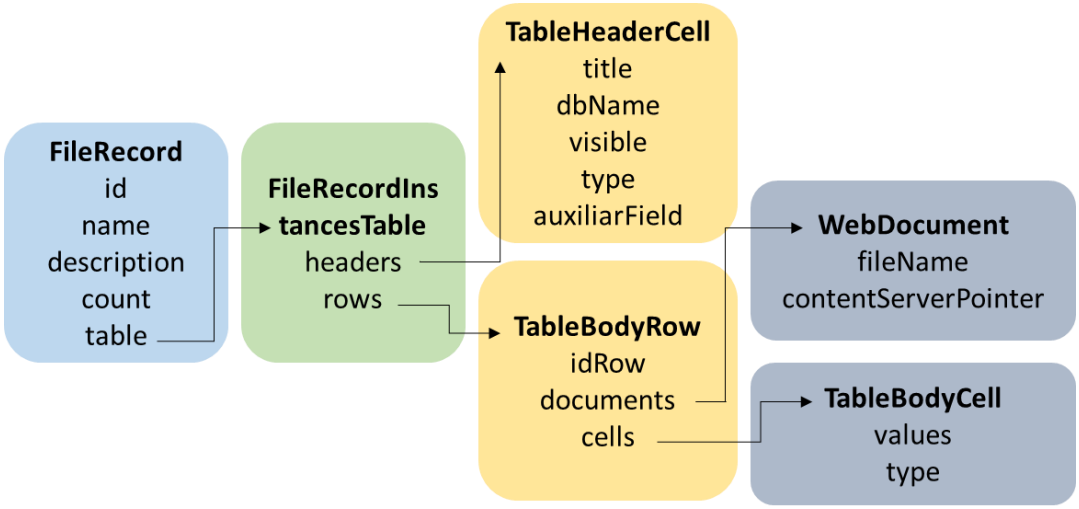


Figura 10: Modelo de datos para la gestión del listado de registros

5.1.2.3 Búsquedas

El módulo *Explorar* proporciona una serie de herramientas de búsqueda muy completas que facilitan al usuario encontrar el/los registro/s deseados. Dado que hay tres tipos de búsqueda, para evitar realizar múltiples llamadas al Servidor y aprovechando el tipado en clases se han creado varias clases para el manejo de información.

La clase principal *SearchFilters* agrupa los tres tipos de búsqueda, incluye un atributo para la *Búsqueda Global*, una lista de búsquedas locales (de tipo *LocalFilter*) y una lista de criterios de búsqueda avanzada (de tipo *AdvancedFilter*). La clase *LocalFilter* contiene el texto, el tipo de filtro y la columna (de tipo *TableHeaderCell*). La clase AdvancedFilter representa un criterio de la búsqueda avanzada y contiene la información de los paréntesis, el operador y un filtro local (de tipo *LocalFilter*).

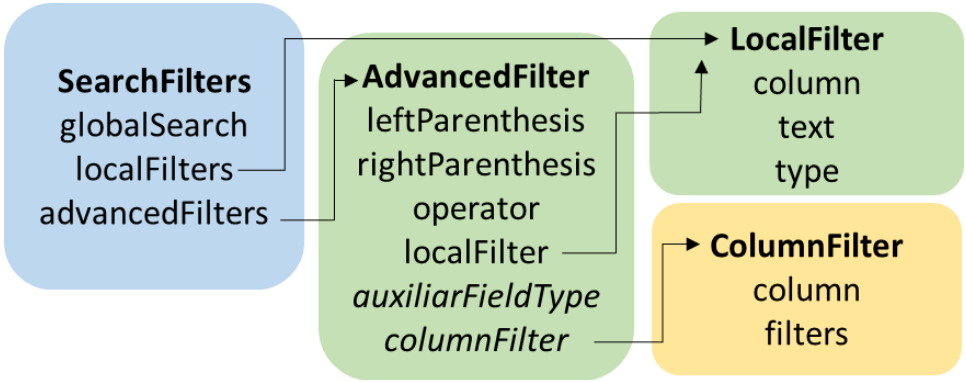


Figura 11: Modelo de datos para la gestión de búsquedas del listado de registros

Además en Cliente también añade un campo auxiliar para la entrada de texto dinámico de los criterios (fecha, asociado, jerárquico, etc) y un objeto *ColumnFilter* necesario para establecer el tipo de filtro que se puede aplicar al criterio. Esta clase *ColumnFilter* contiene la columna (de tipo *TableHeaderCell*) y una lista de filtros posibles.

Aparte de estas clases principales y relaciones, se han creado otras estructuras de datos menos relevantes que se comentan más adelante junto a los procedimientos desarrollados.

5.2 Servidor

En esta subsección se explica cómo se desarrollan las funcionalidades necesarias que realiza el Servidor de la aplicación en lo referente a este TFG y algunos de los problemas encontrados. Toda la implementación lógica del Servidor se ha desarrollado en *C#* y lo referente a bases de datos mediante sentencias *SQL*.

5.2.1 Motor ECM

5.2.1.1 Módulo Administración

Como se ha comentado en secciones anteriores, ha sido necesario incorporar una nueva tabla *PWSAUXILIARES* al *Motor ECM* que contiene información de referencia a las *Tablas Asociadas* (*PWAS**) y *Tablas Jerárquicas* (*PWJS**), por ello desde un gestor de base de datos (*SQL Manager*) se ha ejecutado un *script SQL* creado del ANEXO A.

Por otro lado se añade la clase abstracta *AuxiliarTable* a las librerías PixelwareApi con varios métodos para obtener, crear, modificar o eliminar Tablas Auxiliares. Estos métodos ejecutan sentencias sobre la base de datos agregando los parámetros de manera preparada para evitar la *Inyección Sql.*
Uno de los problemas encontrados ha sido en las sentencias de creación de *Tablas Auxiliares,* donde se le permite al usuario definir el nombre/referencia de la tabla y dado que el nombre de una tabla SQL no se puede fijar como una sentencia preparada había una vulnerabilidad.
La solución ha sido filtrar el texto que nos llega del Cliente para el nombre de la tabla con una expresión regular antes de ejecutar la sentencia, de esta manera se evitan posibles ataques descartando cualquier texto que no cumpla el formato válido para el nombre de una tabla (primer carácter solo letras y resto solo letras o números).

```
Regex rgx = new Regex(@"^[a-zA-ZñÑ][a-zA-Z0-9ñÑ]*$");
if(!rgx.IsMatch(nameTable)){ //Realizar modificación en bd}
```

Por otro lado antes de realizar cualquier sentencia *SQL* de modificación sobre la base de datos se tiene que consultar los permisos que tiene el usuario en cuestión de igual manera que se hace en las clases *AssociatedTable* y *HierarchicalTable* existentes. Se consultan con métodos que ofrece la librería *PixelwareApi* dado el identificador del usuario.

```
bool permission =
       userFileSystem.User.BasePermissions.GetPermission("EliminarValorAsoc");
```

Como se tiene una correlación en *PWSAUXILIARES* con cada la *Tabla Asociada* o *Tabla Jerárquica,* las modificaciones sobre una afectarán a su tabla referenciada. Por ejemplo para eliminar una *Tabla Asociada* se eliminará su valor de *PWSAUXILIARES (*si existe) y a continuación se eliminará la *Tabla Asociada* en cuestión con todos sus valores.

5.2.1.2 Módulo Tareas

Para éste módulo no ha sido necesaria ninguna modificación adicional en el *Motor ECM*, pero si la investigación de funciones de la librería *PixelwareApi* permiten obtener el listado de registros, aplicar filtros a la obtención de registros, consultar o modificar las preferencias del usuario para la *Configuración de Visualización*, obtención de *Valores Actuales* y obtención de documentos.

5.2.2 RESTful WebApi

El servicio *RESTful WebApi* de la aplicación actúa como punto intermedio y se encarga de recibir las peticiones del Cliente, procesar y manipular los datos e interaccionar con el Motor ECM o la caché. Contiene los bloques *Controllers* y *Providers* como se explica en la sección de diseño.

5.2.2.1 Controllers

En este grupo se crean las funciones que reciben las peticiones REST del Cliente, cada función contiene un identificador de ruta URI así como los datos enviados y devueltos. Estos datos son de tipos básicos (string, int, bool, etc) o de clases tipadas definidas en ambos sentidos de la comunicación. Por ejemplo la definición de la función para crear una tabla auxiliar es la siguiente:

```
[Authorize]
[HttpPost]
[Route("createAuxiliarTable/{lengthValue}/")]
public int CreateAuxiliarTable(int lengthValue, WebAuxiliarTable auxiliarTable)
{//Llamar al Provider, devolver los datos }
```

La etiqueta *Authorize* nos permite mantener la información del usuario en *LocalStorage* sin cookies, y que podremos acceder desde la librería *SesionData.* La etiqueta *HttpPost* indica que será llamada desde una petición *POST* del Cliente cuya URI será *createAuxiliarTable/* con los parámetros *lengthValue* y un objeto *WebAuxiliarTable* JSON que es traducido automáticamente a las clases del Servidor como se explica en anteriores secciones. En este ejemplo, se devuelve como respuesta al Cliente un valor de tipo entero (*int*).

Estas funciones instancian y llaman a los *Providers* para realizar las acciones necesarias y obtener los datos que devolver al Cliente.
Se crean dos clases principales, *NodeController.cs* para la administración de *Tablas Auxiliares* del *Módulo Administración* y *FileProvider.cs* para la gestión de información con los registros del *Módulo Explorar.*

5.2.2.2 Providers

En este grupo se desarrolla toda la lógica de la parte *Servidor WebApi*. Ofrece una serie de funcionalidades que son llamadas desde los *Controllers* y que interactual con el Motor ECM o la caché. Se crean dos *Provider* principales, *NodeProvider.cs* para la administración de *Tablas Auxiliares* y *FileProvider*.cs para la gestión de información con los registros del *Módulo Explorar.*

NodeProvider.cs

Se crean los métodos necesarios para obtener, crear, editar o eliminar *Tablas Auxiliares*. Estos métodos interactúan con la clase *AuxiliarTable* del *Motor ECM* antes explicada, donde se llaman a sus métodos de manera estática ya que es abstracta. En caso de solicitar las *Tablas Auxiliares*, se transforma los datos obtenidos a la clase *WebAuxiliarTable* para su envío al Cliente. Como no se requiere paginación en el listado de *Tablas Auxiliares*, se aprovecha la información para realizar las búsquedas en Cliente con filtros mediante *AngularJS*. En caso de creación o eliminación de una nueva *Tabla Auxiliar*, se añade/elimina en la base de datos usando las librerías del *Motor ECM* como se explica anteriormente, y además es necesario realizar dichas modificaciones en la cache mediante las funciones que ofrece la librería *MemoryCache*.

```
UserFileSystem userFileSystem = SessionData.FileSystem.Login(idUser);
// Elimino la tabla de la base de datos
int result = AuxiliarTable.DeleteAuxiliarTable(auxiliarTable.Reference,
      (AuxiliarTableType)auxiliarTable.Type, userFileSystem);
// Elimino la tabla de la cache
if (result >= 0){MemoryCacheManager.DeleteValue(auxiliarTable.Reference);}
```

Para obtener el listado de valores asociados o jerárquicos se usan funciones de *MemoryCache*, que nos devuelven los datos en formato *AuxiliarValuesList.*
Se recuerda que los valores asociados se mostran en Cliente de forma paginada y los jerárquicos en forma de árbol, por esta razón se realiza una serie de procesos para obtener el formato adecuado a devolver (*AssociatesAuxiliarTableResult* para los asociados o *HierarchicalAuxiliarTableResult* para los jerárquicos).

Para los valores asociados, la lista que contiene la clase a devolver *AssociatesAuxiliarTableResult* contiene el total de valores asociados (necesario para establecer el número de páginas) y una lista con solo los valores (*AuxiliarTableValue*) de la página visualizada por el Cliente para disminuir la carga de datos que devuelve el Servidor. Es decir, si se muestran 12 valores por página y se solicita la página 3, solo se devolve los valores del número 24 al número 36. Por otro lado también se tiene en cuenta la posible búsqueda que haya efectuado el usuario, por tanto se realiza un previo filtro con expresiones *labda* mediante la función *FindAll* de la librería LINQ de C# antes de obtener el listado de valores paginado.

```
filteredValues= Values.FindAll(x => x.Value.Contains(filter));
```

Para los valores jerárquicos, la lista que contiene la clase a devolver *HierarchicalAuxiliarTableResult* contiene el total de valores jerárquicos y una lista con valores nodo del árbol (*CommonTreeNode*). Dicho árbol se crea desde la lista de valores que se obtiene de la caché y se forma usando los identificadores (*id*) e identificadores padre (*parentId*) de cada valor para formar los nodos del árbol agrupando padres e hijos. Al igual que se realiza la paginación para los valores asociados, los árboles también requieren de optimización para no devolver la estructura completa de cientos de nodos del árbol. Por esta razón inicialmente solo se devolven los nodos padre del árbol cuyo identificador del padre (*parentId)* sea 0, es decir, los que están en la cima del árbol. En peticiones posteriores solo se devuelven los nodos hijo de un nodo padre que se quiera expandir mediante la especificación del *id* del padre que será igual al *parentId* de los hijos.

Pongamos que se tiene en Cliente un árbol inicial con 3 nodos sin expandir *nodo1*, *nodo2* y *nodo3* cuyos *id* son 1, 2 y 3 respectivamente. Si el usuario solicita la expansión del *nodo2*, enviará al Servidor la petición solicitando la expansión del *nodo2* cuyo *id* es 2 y por tanto devolverá al Cliente solo los nodos hijo cuyo *parentId* sea 2. De esta manera se optimiza en envío evitando redundancia de datos entre el Cliente y Servidor.

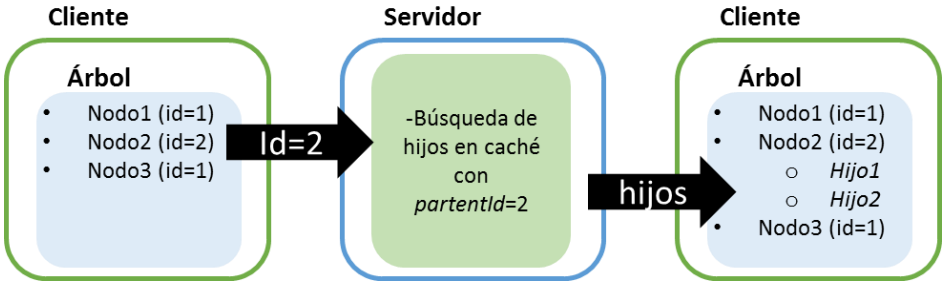


Figura 12: Expansión de nodos del árbol de valores jerárquicos

Por otro lado las búsquedas tienen un comportamiento peculiar recomendado por el tutor para facilitar la tarea al usuario. Tras buscar un valor en Cliente, el árbol expande automáticamente todos los nodos que es necesario expandir hasta llegar al nodo resultado de la búsqueda; en caso de haber más de un resultado, se informa al usuario y este podrá iterar entre los distintos resultados, que expandirán de la misma manera los nodos necesarios hasta llegar a cada nodo resultado.

Por ello, en vez de devolver los nodos hijo dado su *parentId* como en el caso anterior, se procede de otro modo:

1. Se filtra el listado de posibles resultados de la búsqueda en lista de valores de la caché usando una *expresión labda*.
2. Inicialmente con el primer resultado de la búsqueda se crea y devuelve un árbol desde el nodo encontrado hasta el padre de la cima del árbol (*root*), además también se indica el número de resultados de la búsqueda. Con esto se simula en Cliente todo el despliegue que sería necesario realizar para encontrar el nodo y el total de resultados.
3. Si el Cliente solicita el siguiente resultado de la búsqueda, se repite la operación pero creando el árbol a partir del siguiente elemento de la búsqueda del paso 1.

Por último el proceso de creación, edición o eliminación de valores asociados o jerárquicos es más complejo en comparación con las *Tablas Auxiliares*, ya que se tiene en cuenta la información actualizada en caché y en base de datos. Primero se realiza la incorporación, modificación o eliminación de valores sobre la base de datos con las librerías del Motor ECM haciendo uso de los métodos de clases *AssociatedTable* o *HierarchicalTable;* si el resultado ha sido correcto se actualiza solo la tabla que contiene dichos valores en caché mediante la clase proporcionada *MemoryCache*, en vez de refrescar la caché al completo.

FileProvider.cs

Se han creado los métodos necesarios para obtener los registros en forma de *grid* de tablas con una estructura de cabeceras, filas y celdas que contiene la clase *FileRecord* anteriormente explicada. Mediante atributos de control (necesarios para interaccionar con el Motor ECM) referentes a la información acerca del usuario en cuestión (permisos, configuraciones, ect) y a la *Tabla de Usuario* que se quiere consultar (índice, esquema,

etc), se accede a la base de datos mediante las librerías del Motor ECM, las cuales nos proporcionan el listado de registros en formato lista de objetos de clase *Record*. Dicha clase de las librerías *PixelwareApi* es muy extensa y representa toda la información de un registro, por esta razón se construye la clase propia *FileRecord* a partir de la lista. Una *Tabla de Usuario* puede tener miles de registros, los cuales a su vez pueden tener decenas de campos en los peores casos, por tanto no es óptimo obtener el listado completo de registros si luego en Cliente no se va a mostrar toda la información.

Se tiene en cuenta varias características:

1. ***Configuración de Visualización* del usuario:** por ejemplo, si para una determinada *Tabla de Usuario* de 40 campos está configurada para visualizar solo 4 campos específicos, no es necesario ni eficiente devolver todos los campos al Cliente.
2. **Paginación:** al igual que al procesar el listado de valores asociados de una *Tabla Asociada* comentado en *NodeProvider.cs*, sólo será necesario devolver la información de la página que se requiere mostrar en Cliente.
3. **Búsquedas:** el usuario aplicará varios filtros de búsqueda con las tres herramientas que tiene a su disposición (búsqueda local, global y avanzada) por lo que se hace una selección de registros que cumplen la búsqueda antes de devolver la información.

La librería *PixelwareApi* del *Motor ECM* cuenta con varios proveedores que proporcionan el listado de registros teniendo en cuenta múltiples variables como relaciones entre tablas, parámetros de control, campos requeridos, condiciones de campos, número de resultados, etc. En concreto se usa el proveedor *PartialDataRecordProvider* que nos permite fijar varios parámetros antes de obtener los registros además de parámetros de control del *Motor ECM*.

Se explica a continuación como se establecen los parámetros del proveedor a partir de las características anteriormente descritas:

1. ***Configuración de Visualización* del usuario:** Se especifica el listado de campos que contiene la *Configuración de Visualización* mediante un listado de tipo *Predicate* de *C#*. Esta configuración se obtiene mediante la clase *GridViewColumnPreferences* de la librería *PixelwareApi* especificando parámetros de control necesarios para el Motor ECM (información del usuario y de la tabla). Dicha clase nos proporciona los campos incluidos que tiene configurados un determinado usuario en una determinada tabla desde la base de datos.
2. **Paginación:** Se especifica el número de registro a partir del cual queremos la lista y cuantos elementos, con esto se nos permite fijar el rango de resultados para obtener los registros contenidos en una determinada página. Dado que la información se muestra paginada en Cliente, si por ejemplo solicita la página 3 con un total de 12 resultados por página, le indicamos al proveedor que el inicio del rango será del registro número 24 al 36
3. **Búsquedas:** Por último especificaremos los filtros que aplicará el proveedor para obtener los registros. Estos filtros se crean a partir de las búsquedas que realiza el usuario en la parte Cliente. El proveedor permite la especificación de filtros de clase *FieldFilterData* de la librería *PixelwareApi*. En la creación del filtro se especifica el campo de la tabla, texto y tipo de operador (contiene, no contiene, es igual, etc), esta información se obtiene del Cliente con la clase propia *SearchFilters* antes explicada. Además se permite fijar la relación entre filtros especificando el operador lógico (*AND* u *OR*) y el agrupamiento de bloques de filtros. De esta forma

se agrupan los tres tipos de búsqueda en grupos independientes con relación lógica *AND*. Cada grupo contiene lo siguiente:

a. **Búsqueda Local:** Se añade un *FieldFilterData* al grupo por cada *LocalFilter* recibido del Cliente (transformando la información de una clase a otra). Todos los filtros tienen una relación lógica *AND*. Por tanto se filtran los registros que cumplan el conjunto de filtros especificado en cada campo.
b. **Búsqueda Global:** Dado un texto de búsqueda global, se crea un *FieldFilterData* por cada campo que tenga la tabla (incluidos y excluidos). Todos los filtros tendrán una relación *OR* y el operador Contiene; se filtran los registros que para alguno de sus campos contengan el texto.
c. **Búsqueda Avanzada:** Se añade un *FieldFilterData* al grupo por cada *AdvancedFilter* recibido del Cliente (transformando la información de una clase a otra). Cada filtro tendrá la relación lógica especificada en *AdvancedFilter*; se filtra los registros que cumplan el conjunto de filtros especificado en cada criterio de la búsqueda avanzada.

Para editar o restablecer la *Configuración de Visualización* también se hace uso de la clase *GridViewColumnPreferences* de la librería PixelwareApi. Estas opciones se gestionan enviando al Cliente una nueva clase creada *TableHeaderConfiguration* que incluye un listado de campos incluidos y excluidos.

Otra de las herramientas que nos proporciona el proveedor de registros *PartialDataRecordProvider* son los valores actuales para un campo y tabla determinados. Los valores actuales son los distintos valores existentes en un campo para una tabla determinada. Por ejemplo para una determinada *Tabla de Usuario* que contiene un determinado campo "estado" (valores como *Aceptada, No aceptada, Rechazada, etc*,) el usuario puede establecer el texto del filtro de búsqueda local a partir de una lista de distintos elementos posibles en la tabla para dicho campo, lo que equivale a la expresión *DISTINCT* en *SQL*. Cuando el Cliente solicite visualizar el seleccionable de *Valores Actuales* de un campo, se obtendrán mediante funciones del proveedor especificando los parámetros de control necesarios como la información del campo y tabla en cuestión. El listado de valores se devuelve al Cliente con una nueva clase creada *CurrentValues* para facilitar su representación posteriormente.

Por último, los documentos asociados a un registro se incluyen en la clase *FileRecord* devuelta al Cliente y especifican un nombre y puntero asociado a un documento del Servidor de Contenidos *ContentServer* del *Motor ECM*. Posteriormente cuando un Cliente solicita la descarga de un documento, se ha creado funciones para acceder a *ContentServer* mediante las librerías *PixelwareApi* obteniendo el *array* de *bytes* que representa el documento. Este *array* se incluye un objeto *HttpResponseMessage* especificando sus cabeceras y formato del adjunto (pdf, Word, etc) para enviárselo al Cliente donde se realizará la descarga, como se explica más adelante,.

```
string contentType = GetMimeType(Path.GetExtension(document.FileName));
var result = new HttpResponseMessage(HttpStatusCode.OK) { Content = new
ByteArrayContent(bytes) };
result.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);
result.Content.Headers.ContentDisposition = new
ContentDispositionHeaderValue("attachment"){FileName = document.FileName};
```

5.3 Cliente

En esta subsección se explica cómo se ha desarrollan las funcionalidades necesarias que realiza el Cliente de la aplicación en lo referente a este TFG y algunos de los problemas encontrados. Toda la implementación lógica del Cliente se ha desarrollado en *AngularJS* tipado con *TypeScript* y lo referente a la parte visual con *HTML5, CSS3, Bootstrap, iconos de FontAwsome* y directivas de *AngularJS*.
Principalmente toda la lógica está en torno a los *Services, Controllers y Templates (Vistas)*. Esta filosofía de modulación de *AngularJS* permite tener una organización de funcionalidades.

5.3.1 Configuración y Rutas

Se tiene dos ficheros de configuración y rutas (*Config.ts* y *Application.ts*) para establecer la organización de ficheros y dependencias y rutas de la aplicación.
El fichero *Application.ts* contiene la definición de *plugins* externos usados en la aplicación (*ui.bootstrap, LocalStorage, ui.tree, etc*) y los identificadores definidos para los *Services* y *Controllers* propios creados. Por tanto tras la creación de un *Service* o *Controller* (tipados en clases por *Typescript*) se añade a este fichero su definición de la forma:

```
.controller(identificadorController1, ClaseController1)
.service(identificadorService1, ClaseService1)
```

El fichero Config.ts contiene además de parámetros de inicialización de dependencias, el enrutado que seguirá la aplicación. Dicho enrutado es necesario en *AngularJS*, ya que la página no se recarga pero su contenido cambia dinámicamente modificando el conjunto de *Templates (HTML)* que se muestran. El sistema de rutas nos lo proporciona *$stateProvider* cuyas definiciones son de la forma:

```
.state('identificadorEstado1', {
       url: "/urlEstado1?parametro1&parametro2",
       templateUrl: "application/templates/templateEstado1.html",
       controller: "identificadorController1"
})
```

Una vez establecido el enrutado de un estado en *Config.ts*, podemos cambiar el contenido mostrado (cambiar a otro estado) desde cualquier punto mediante

```
$state.go('identificadorEstado1'{parametro1: "hola", parametro2: "adios"});
```

5.3.2 Service

Los *Service* de *AngularJS* son módulos creados la obtención de información y comunicación con otros servicios mediante inyecciones de dependencia. El *framework* ofrece servicios propios como $*http* (*ng. IHttpService*) que proporciona los métodos necesarios para realizar peticiones http *AJAX* a Servidores web. Por otro lado también se sigue esta filosofía (módulo para la obtención de información) para crear nuestros propios servicios y usarlos en la aplicación. Se crean dos nuevos servicios, *NodeService.ts* y *FileService.ts* que inyectan la dependencia del servicio *$http* para realizar las peticiones

Http al Servidor *RESTful WebApi*. Dado que son nuevos ficheros, se añaden sus identificadores en *Application.ts*. Estos ficheros contienen las funciones necesarias para realizar las llamadas a cada función de los *Controller* del Servidor (*NodeController.cs* y *FileController.cs*) y para ello se especifican sus determinadas *URI* y parámetros en la llamada. Por ejemplo la función del servicio encargada de la creación de tablas auxiliares (mismo ejemplo el explicado en los *Controller* del Servidor *RESTful* pero desde la parte Cliente), realiza la siguiente llamada al Servidor mediante una petición http.

```
this.$http.post('api/nodes/createAuxiliarTable/'+lengthValue,
       JSON.stringify(auxiliarTable),{headers:{'Content-Type':'application/json'}})
               .then((response) => {// Devolver response al Controller
```

Se realiza un *POST* a la *URI* del Servidor *RESTful WebApi* "*api/nodes/createAuxiliarTable/*" indicando los parámetros *lengthValue* y un objeto *WebAuxiliarTable* serializado en *JSON*.

5.3.3 Template-Controller

La aplicación mostrada al usuario contiene una serie de paneles contenedores (menús, contenido, etc) que cambian dinámicamente según las rutas llamadas, cada bloque está formado por un *Template (html)* y su *Controller* definido en *Config.ts* que se muestran en el navegador según la estructura jerárquica fijada mediante *ui-view* de Angular. El proyecto tiene multitud de *Templates* y *Controllers,* por lo que se describen solo los desarrollados en este TFG a rasgos generales para no extenderse.

Cada *Controller* gestiona los datos y se comunica con los *Service* explicados anteriormente para poder realizar todas las funcionalidades requeridas. Se guarda el contenido de los datos de forma tipada en clases en los atributos del contexto *$scope* (único para cada *Controller*). Dichos atributos son necesarios para el manejo de los datos tanto en el propio *Controller* como en el *Template.*

Cada *Template* muestra la visualización del bloque en el navegador y mediante directivas de *AngularJS* se representa la información del modelo de datos y permite la interacción con el *Controller.* Para mantener la aplicación *responsive* se usa el sistema de columnas *grid* de *Bootstrap* definiendo a cada bloque el espacio disponible (hasta el sumatorio de 12 columnas), con lo que se adapta a las distintas pantallas.

La aplicación tiene en todo momento un menú en la parte superior (*Template* y *Controller* ya desarrollado) para la navegación entre módulos de la aplicación (*Administración, Explorar y Tareas*).

5.3.3.1 Módulo Administración

El *Módulo Administración* ofrece en todo momento un menú lateral para la navegación entre las distintas opciones de *Administración* superior (*Template* y *Controller* ya desarrollado). Una de las opciones es la gestión de *Tablas Auxiliares* donde al hacer clic el enrutado muestra el contenedor con toda la gestión de *Tablas Auxiliares*. Por ello se crea en la capeta *Administración* un nuevo *Controller ManageAuxiliarTablesController* y *Template ManageAuxiliarTables.html.* Dicho *Template* principal tiene a su vez otros *Template* en su interior.

Listado de Tablas Auxiliares

El *Controller* obtiene del *Service (NodeService.ts)* el listado de *Tablas Auxiliares* que será guardado en un atributo de *$scope* (de clase *WebAuxiliarTable*) y representado mediante el *Template* con la directiva *ng-repeat* mostrando además un *Tooltip* para ayudar al usuario.
Para fijar la visualización en forma de *scroll* con un alto fijo adaptable al alto disponible se usa la clase *CSS* creada *scrollable-modal-header-content* que hace uso de la propiedad *vh.*

```
<div class="scrollable-modal-header-content">
<a ng-repeat="row in auxiliarTablesSearch =
       (auxiliarTables | filter:{ title:auxiliarTableSearch })| orderBy:'title'"
       data-ng-click="selectAuxiliarTable(row)" class="list-group-item"
       ng-class="{'active': row.reference==selectedAuxiliarTable.reference}"
       uib-tooltip="{{row.description}}">{{row.title}}</a>
</div>
```

Se guarda la tabla seleccionada en $*scope* mediante la directiva *ng-click*, que se utiliza para realizar modificaciones sobre la tabla (Editar o Eliminar) así como para obtener el listado de valores.
Dado que se tiene en un atributo de *$scope* todo el listado de tablas, la búsqueda se realiza a través de un filtro de Angular especificando como parámetro *filter* de *ng-repeat* el valor introducido en el input de búsqueda.

Listado de Valores

Una vez seleccionada una *Tabla Auxiliar*, el *Controller* obtiene del *Service (NodeService.ts)* el listado de *Valores Asociados* o *Jerárquicos* (según el tipo de tabla) se guarda en un atributo de *$scope* (de clase *AssociatedAuxiliarTableResult* o *HierarchicalTableResult*) y representa mediante el *Template* con la directiva *ng-repeat*. Al seleccionar un valor, se guarda en $*scope* mediante la directiva *ng-click*, que será utilizado para poder realizar modificaciones del valor (Editar o Eliminar).

Asociado
En el caso de valores asociados, la forma de representación es una lista paginada, por ello se usa de la directiva *pagination-control* desarrollada para la selección de página que se pedirá al *Service*. Esta directiva proporciona un *Template* para la visualización del menú de paginación y mediante la definición de atributos en *$scope* se controla la página requerida escuchando mediante *$watch* de *AngularJS*, que enviaremos a *Service* en las peticiones de valores. Esta propiedad nos permite realizar operaciones en los *Controller* cuando un atributo de *$scope* cambia de valor.

```
$scope.$watch("atributoEscuchado", function (newValue, oldValue) {//Acciones
```

Para las búsquedas, el valor introducido se envía a *Service* con las peticiones de valores para obtener los resultados de la búsqueda en forma paginada.

Jerárquicos
En el caso de valores jerárquicos, la forma de representación es un árbol, que como se explica en la parte del servidor *RESTful*, se irá completando mediante peticiones a *Service* a medida que el usuario desea expandir nodos. De esta forma, la carga del árbol es más óptima evitando traer todos los nodos del árbol, pero más difícil de programar en Cliente. Cuando el usuario expanda un nodo, se envía a *Service* tal petición indicando el

identificador del nodo a expandir, y tras la respuesta con los nodos hijos se fijan al padre desde el *Controller*. Para la representación se usa el *plugin ui-tree* (además de la directiva *ng-repeat*) estableciendo nuevos *Template* mediante la directiva *ng-include* que definen la visualización y el comportamiento de un nodo padre o un nodo hijo.

```
<div data-ui-tree><ol data-ng-model="hierarchicalValues.values">
      <li data-ng-repeat="node in hierarchicalValuesDialog.values |
                          orderBy:'title'" data-ui-tree-node
          data-ng-include="node.hasChilds ? 'template_tree_parent_nodes' :
                          'template_tree_terminal_node'">
</li></ol></div>
```

Por otro lado, las búsquedas en árboles son algo peculiares. Tras enviar la petición de búsqueda, el *Service* devuelve la ruta desde el resultado hasta el padre de la cima del árbol (root) con todos los hermanos del nodo y para poder representarlo, el *Controller* se encarga mediante algoritmos recursivos de simular tal expansión de nodos y selección del resultado. Se puede contemplar el algoritmo creado en el ANEXO B.

En la siguiente captura para la Tabla Auxiliar de Provincias, se efectúa la búsqueda "andorra" sobre el listado de sus valores jerárquicos en forma de árbol. Se puede apreciar cómo se despliega el árbol para mostrar el resultado al usuario.

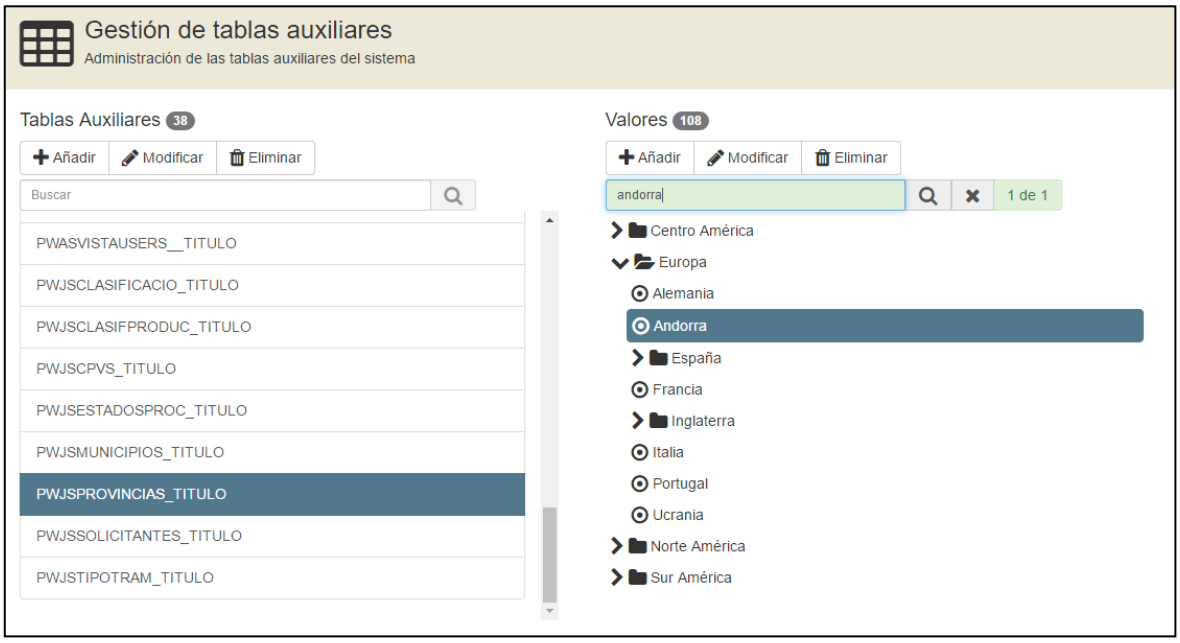


Figura 13: Vista Gestión de tablas auxiliares, búsqueda de valores jerárquicos en el árbol

Por último, los menús de cada listado de la página, ofrecen las funcionalidades para crear, editar o eliminar *Tablas Auxiliares* o *Valores* (habilitados según los permisos del usuario). Dichas funcionalidades (con un valor seleccionado previamente) abren un diálogo correspondiente mediante el *plugin ng.Dialog* desde el *Controller* en el que se especifica el *Template* que define la visualización y comportamiento del formulario contenido. Estos formularios permiten al usuario rellenar la información de cada caso para posteriormente desde el *Controller* realizar la petición al *Service*. Cada diálogo muestra avisos de advertencia así como validaciones dinámicas de los campos. El resultado se informará al

usuario, y en caso de error no cerrará el diálogo para poder corregir los errores. La siguiente captura muestra el diálogo de *Edición de una Tabla Auxiliar*.

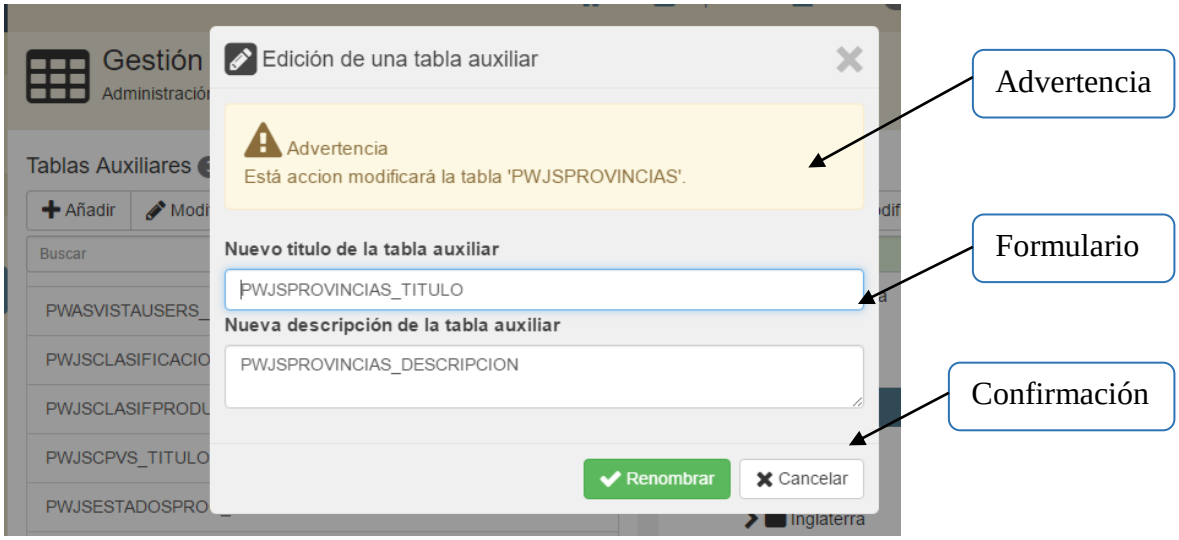


Figura 14: Vista diálogo creación de tabla auxiliar

5.3.3.2 Módulo Explorar

El *Módulo Explorar* ofrece en todo momento un menú lateral para la selección de Tablas de Usuario y por otro lado un contenedor donde se muestra el listado de registros de la *Tabla de Usuario* seleccionada en forma de tabla. Por ello se crea en la carpeta Explorar para el menú lateral el *Controller SrchApplicationController* y *Template Application.html*, y para la visualización de registros el *Controller RecordsController* y *Template Records.html*. Dichos *Template* tienen a su vez otros *Template* en su interior. También se crean otros *Template* de menor complejidad para informaciones al usuario llamados desde las rutas.

Menú lateral de selección de *Tablas de Usuario*

El *Controller* del menú lateral (*SrchApplicationController*), obtiene del *Service* un listado de *CommonTreeNode* con las *Tablas de Usuario*. En este caso el árbol se devuelve completo debido a su reducido tamaño en comparación con los árboles de *Valores Jerárquicos.* El sistema de representación es el mismo que en los Valores Jerárquicos mediante *ui-tree* y nuevos *Template.* Ofrece un sistema de búsquedas similar al de los árboles de *Valores Jerárquicos* simulando las expansiones necesarias hasta llegar a los resultados.
Por último una vez seleccionado un nodo (*Tabla de Usuario*) se llamará al enrutado necesario (*$state.go*) para cambiar el estado y representar el listado de registros mediante el *Controller RecordsController.*

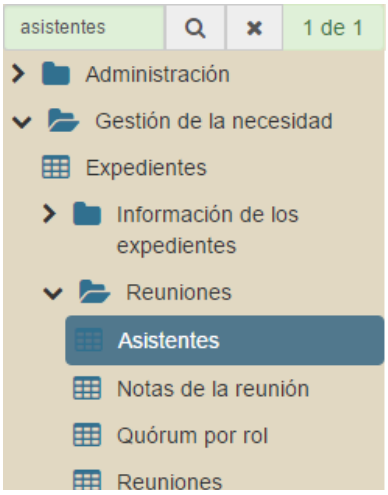


Figura 15: Vista menú de selección de Tablas de Usuario

Visualización de Listado de Registros

El *Controller* obtiene el listado de registros del *Service* en formato *FileRecords* y los representa mediante *ng-repeat* en forma de tabla además de otras directivas. El formato es paginado por lo que se ofrece las herramientas de selección de página mediante la directiva *pagination-control.* Cada cambio de página requiere una nueva solicitud de valores al *Service.*
Para visualizar el resto de campos (*campos no visibles*) de cada registro, el usuario puede abrir un diálogo para cada registro que ofrece un listado de cada *campo no visible* y su valor. Este listado de datos se obtiene mediante solicitudes al *Service* que devolverá la información tipada en la clase *FileRecordInstancesTable.*
Otra de las opciones de la ventana es la *Configuración de Visualización* que muestra un diálogo (*ng.Dialog*) con la configuración actual y la posibilidad de modificarla o reestablecerla. Los datos se obtienen y envían mediante solicitudes al *Service* y se realiza mediante el tipado en la clase *TableHeaderConfiguration*. Una vez editada la configuración, el *Service* nos devolverá en las peticiones del listado de registros, los cambios establecidos con la nueva configuración.

Figura 16: Vista diálogo Configuración de Visualización

Descarga de Documentos

Los registros de la tabla que tengan asociado documentos (*WebDocument*) representan en el *grid* un seleccionable de documentos donde el usuario podrá descargarlos. Tras la selección, se le solicita al *Service* tal documento que además incluirá en la petición el valor *arraybuffer* para el parámetro *responseType*. Una vez devuelto el documento se procede a la descarga automática del mismo, creando en el *Controller* un elemento dinámicamente y asignándole el *array* de bytes con *Blob*.

```
var file = new Blob([data], {type: contentType});
var fileURL = window.URL.createObjectURL(file);
var anchor = angular.element('<a/>');
anchor.attr({
      href: fileURL, target: '_blank', download: document.fileName})[0].click();}
```

Búsquedas

El usuario puede utilizar los tres tipos de búsqueda disponible, a la hora de obtener el listado de registros, se envía al *Service* los datos de búsqueda en la clase tipada *SearchFilters*. Esta clase se crea mediante la información que el usuario introduce en las búsquedas, donde posteriormente el *Controller* se encarga de moldearla para enviarla.
La Búsqueda Local ofrece un listado de filtros (contiene, no contiene, es igual, etc) para a aplicar a un campo que se calculan dinámicamente en el *Controller* según el tipo de campo y se traducen acorde a los ficheros de traducción usando el *Service $translate*. Dicho listado se gestiona en el *Controller* mediante la clase *ColumnFilter*. Como se ha comentado en anteriores secciones, uno de los filtros es *ValoresActuales*. Dicho filtro abre un diálogo con un listado de posibles valores seleccionables para el texto de la búsqueda local del campo en cuestión. El listado de valores se obtiene de una petición al *Service* que nos devuelve la información en la clase tipada *CurrentValues*. La visualización del listado seleccionable de filtros se realiza mediante una lista desplegable usando clases de *Bootstrap* y directivas de *Angular*.

```
<a class="input-group-addon dropdown-toggle" data-toggle="dropdown"
      <i class="fa fa-filter"></i></a>
<ul class="dropdown-menu">
      <li data-ng-repeat="filter in columnsFilters[$index].filters track by $index"
          data-ng-class="{'active':localFilters[$parent.$index].type==$index}">
      <a data-ng-click="setColumnFilterType($parent.$index,$index)">
          {{filter | translate}}</a></li></ul>
```

Finalmente el *Controller* gestiona un listado de búsquedas locales realizadas para cada campo especificando una lista de clase *LocalFilter* en la clase *SearchFilters*.

Búsqueda Avanzada

La búsqueda avanzada ha sido una de las tareas más complejas en el desarrollo del TFG debido a que cada criterio añadido no puede interpretarse simplemente como un *LocalFilter* de una *Búsqueda Local*. Cada criterio se compone principalmente de los mismos atributos que un *LocalFilter* (filtro, texto, campo) además del operador lógico (*AND* u *OR*) y los paréntesis, pero la forma de introducir el texto es variable según el tipo de campo seleccionado como se ha explicado en secciones anteriores. De esta forma si se selecciona un campo de tipo fecha para un criterio, la forma de introducir el texto del filtro será un calendario, si es de tipo *Real* tendrá que comprobar el formato del número, si es de tipo asociado o jerárquico tendrá que abrir un diálogo para seleccionar el/los valor/es deseados (de forma paginada o en árbol igual que en el módulo de Administración), etc. Por ello se crean *Template* de visualización y comportamiento para cada tipo de campo que serán añadidos según la selección del campo y gestionados por el *Controller*. Se han usado clases y componentes de *Bootstrap* para mejorar la visualización de los *Template* como por ejempo *uib-datepicker-popup* en los calendarios.

Otro de los problemas encontrados es la posibilidad que se ofrece al usuario de modificar el orden de los criterios creados. Por esta razón se almacena toda la información referente a un criterio como los listados que necesita en la clase *AdvancedFilter*, para poder cambiar el orden desde el *Controller* sin perder la correlación de los datos. Las validaciones se han gestionado con el atributo *auxiliarFieldType* usado para manejar la información necesaria para validar datos (valores máximos, formato de fechas, etc) o la gestión de etiquetas de valores asociados o jerárquicos tras seleccionar elementos en sus diálogos. Aunque el

atributo *auxiliarFieldType* no tenga tipo definido, la información en él se gestiona siempre con clases tipadas creadas como *DateFieldData TextFieldData IntegerFieldData RealFieldData.*

Figura 17: Vista Búsqueda Avanzada

Finalmente el *Controller* obtiene un listado de criterios creados especificando una lista de clase *AdvancedFilter* en la clase clase *SearchFilters*.

La búsqueda global se recoge del input básico que ofrece la página y también se añade a la clase *SearchFilters* para unificar los datos y realizar menos llamadas al *Service.*

Una vez devuelto el listado de registros con los filtros aplicados se presentan de la misma forma paginada y se mantiene toda la información de las tres herramientas de búsqueda para permitir al usuario la edición de la misma en posteriores búsquedas.

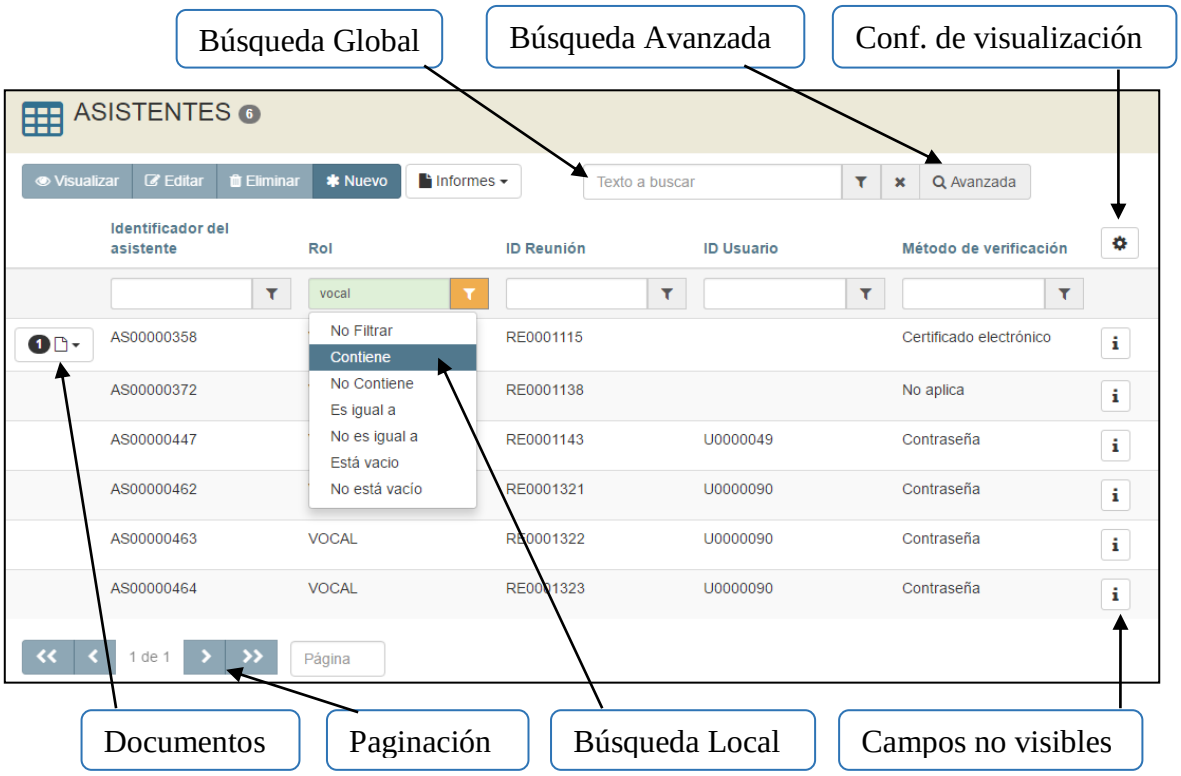


Figura 18: Vista listado de registros del Módulo Explorar

Por último el menú superior de ventana (Visualizar, Editar, Eliminar, Nuevo e Informes) se realiza mediante formularios dinámicos creados en Servidor que aún están en desarrollo por otros desarrolladores del proyecto. Por ello solo se ha integrado la funcionalidad con la opción *Visualizar* cuyos servicio de creación de formularios está creado.

Visualización de Formularios de un Registro

Aunque no se contempla en los requisitos funcionales, se ha desarrollado la visualización de formularios de un registro mediante la opción *Visualizar* para completar la funcionalidad básica del módulo *Explorar* y poder realizar pruebas concluyentes en la sección de pruebas.
Se crean dos nuevos *Controller (RecordFormMenuController* y *RecordFormController)* y sus *Template* básicos para la visualización de un nuevo menú lateral de opciones y una superficie de contenido donde se muestran los formularios. El *Controller RecordFormController* llama al *Service FormService* del que obtiene los formularios y el contenido de sus datos con lo que representa en la página la información.

6 Integración y pruebas

Una vez desarrollado cada módulo de la aplicación, se realizan varias pruebas para comprobar el funcionamiento correcto, las mejoras con respecto al software obsoleto y la integración con el resto de módulos del proyecto general.
Se realiza una serie de pruebas sobre cada módulo desarrollado para comprobar su comportamiento antes de pasar a la siguiente fase de desarrollo.

6.1 Pruebas unitarias

Para cada función de cada módulo diferenciado de la aplicación se comprueba el comportamiento frente al paso de argumentos válidos e inválidos y se analizan los resultados devueltos por cada función. A menos que un parámetro sea de tipo objeto (o *any* en *Typescript*), siempre se manejan datos tipados en clases.

6.2 Pruebas de integración

Dada la gran modularidad de funcionalidades en la aplicación, se comprueba el envío de datos en la comunicación entre distintas capas y entre distintos agentes (Cliente-Servidor).
En Cliente se comprueba el funcionamiento correcto entre módulos mediante las inyecciones de dependencia de Angular donde gracias a *Typescript* se evitan errores en el envío de datos de un módulo a otro.
En Servidor se controlan las posibles excepciones en las comunicaciones con el Motor ECM y el tipo de datos manejado.
Por otro lado entre Cliente y Servidor se comprueba que las traducciones de *serialización* JSON de las peticiones REST se realizan de forma correcta, comprobando los datos obtenidos en la comunicación, por ello es imprescindible que las clases del Servidor tengan las mismas etiquetas de traducción que el nombre de atributos en Cliente.
Por último como se ha comentado al final de la sección de desarrollo, se ha probado la integración con otras funcionalidades desarrolladas por otros desarrolladores. En concreto la visualización de formularios de un registro donde se usa un *Service* y datos diferentes a los desarrollados.

6.3 Pruebas de Regresión y Validación

Debido al desarrollo en paralelo del proyecto entre varios desarrolladores, antes de realizar cambios importantes que puedan afectar al comportamiento de otros módulos de la aplicación, se realizan pruebas de regresión en posibles zonas de conflicto.

Una vez finalizado el desarrollo de cada módulo, se comprueba el cumplimiento de los requisitos funcionales (RF) y no funcionales (RNF) definidos en la fase de análisis.

6.4 Pruebas de sistema

En las pruebas de sistema se comprueban varios factores de la aplicación:

- **Navegación:** Se comprueba que el usuario puede acceder a cada funcionalidad de la aplicación en pocos clic.
- **Adaptabilidad:** La vista como se comenta en secciones anteriores se ha diseñado y desarrollado de manera *responsive*, por ello se ejecuta la aplicación en distintos

dispositivos y pantallas para comprobar que cada bloque se visualiza correctamente.

- **Recuperación y tolerancia a fallos:** Se comprueba cómo se comporta el sistema ante la entrada de datos errónea (la aplicación valida la información antes de usarla), la realización de acciones sin permiso (la aplicación informa al usuario) o los fallos del Servidor (la aplicación informa del error).
- **Seguridad:** Se comprueba en cada módulo que el usuario está autenticado antes de mostrar cualquier información.

6.4.1 Pruebas de rendimiento

Adicionalmente aunque el proyecto general no se ha completado, se han realizado pruebas de rendimiento en los módulos realizados en este TFG para argumentar las motivaciones iniciales del trabajo.

Se usa la herramienta *Apache JMeter* usada en prácticas de la universidad, para realizar pruebas de carga y medir el rendimiento del Servidor. La finalidad es comparar el software obsoleto con el remodelado con tecnologías modernas para argumentar una de las principales finalidades del proyecto, que es la liberación de carga del Servidor.

Se ha comprobado el rendimiento del Servidor con un mismo escenario para las dos aplicaciones. Por un lado el antiguo *Pixelware Search* y por otro su equivalente remodelado *Módulo Explorar* del nuevo *SolutionsSpa.*

Usando la misma base de datos, el escenario comprende los siguientes pasos:

1. Visualización del listado de registros de la Tabla de Usuario Asistentes.
2. Búsqueda global por “Presidente”
3. Con los resultados obtenidos, visualización de los formularios de un registro.

Para grabar la simulación desde *JMeter* se usa la opción *Servidor Proxy HTTP* indicándoselo al navegador e incorporando las opciones necesarias que incluyen Cookies en *JMeter* para el software antiguo (*HTTP Cookie Manager*) y deshabilitación de los controles de seguridad *Authorize* en el software remodelado (*Módulo Explorar* de *SolutionsSpa).*

La prueba se ha realizado estableciendo como parámetros 100 usuarios con una periodicidad de 3 sec y los datos se han recogido con el *listener* de *JMeter Summary Report.* Se adjuntan capturas de la prueba en el ANEXO B.

Software	Tiempo	Datos enviados	Througput
Search	119 sec	819,19 KB/sec	181,6 operaciones/sec
SolutionsSpa	40 sec	45,17 KB/sec	22,1 operaciones/sec

Tabla 1: Resultados de la prueba de rendimiento JMeter

Tras realizar la prueba de *Stress* con los usuarios simultáneos se comprueba que el nuevo software necesita menos tiempo para realizar la simulación completa (2,98 veces más rápido), realiza menos operaciones por segundo (8,21 veces menos sobrecargado) y mueve una carga de datos mucho menor para mostrar lo mismo (18,13 veces menos datos enviados). Estos resultados son debidos a que la página no se recarga con *AngularJS* y solo se realizan las mínimas peticiones al Servidor para obtener los datos.

Por otro lado como punto desfavorable el Cliente requiere de más recursos para ejecutar toda la lógica traducida a *JavaScript*, por tanto los usuarios requerirán equipos más potentes.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

La participación en el proyecto general *SolutionsSpa* para realizar este TFG ha proporcionado el conocimiento y la aplicación de nuevas tecnologías y herramientas de desarrollo software informático.

Aunque en proyectos de gran escala como este, la misma persona no lleva a cabo múltiples fases del ciclo de vida, se adapta en este trabajo la realización de varias fases para tener una visión real más ampliada. De esta forma se explican y argumentan todas las características que se han realizado.

Debido a la remodelación de un software existente y el uso de tecnologías desconocidas en la formación universitaria, este trabajo ha conllevado varias sesiones de investigación propia para poder realizar todas las funcionalidades además de reuniones con el tutor y desarrolladores de la empresa para resolver dudas.

Una vez completadas todas las fases y ejecutadas pruebas, se puede corroborar las metas definidas en las motivaciones del trabajo, obteniendo un software más eficiente que permitirá mejorar la experiencia del usuario final. Por otro lado los Servidores finales tendrán una mejora notable del rendimiento debido a la reducción de carga de trabajo.

7.2 Trabajo futuro

Como el título indica, el proyecto general *Pixelware SolutionsSpa* se compone del *Desarrollo de Cliente web integrado de gestión de contenidos y de flujos de trabajo sobre los motores de Pixelware Solutions y jBPM,* es un proyecto de investigación de la empresa donde, se ha participado en el desarrollo de una parte del mismo para la realización de este TFG. Por esta razón en el futuro se participará en otros módulos del proyecto general para completarlo en su totalidad.

En este trabajo se han realizado parte de los módulos que componen el *Cliente web* y la *gestión de contenidos* sobre los motores de *Pixelware Solutios* (*Motor ECM*) y en un futuro se participará en la realización de la gestión de *flujos de trabajo* (*Módulo Tareas*) sobre los motores *jBPM.*

JBPM es un motor de flujos de trabajo escrito en *Java* para poder gestionar procesos de negocio y comunicación de usuarios y desarrolladores mediante diagramas gráficos. Al igual que el desarrollo de los módulos de este TFG, dicha tecnología conllevará fases de investigación adicionales y nuevas fases del ciclo de vida software incremental para poder desarrollar las nuevas funcionalidades.

Referencias

[1] Framework AngularJS . [Último acceso 22/05/2016]. URL: *https://angularjs.org/*

[2] Documentación y ejemplos AngularJS. [Último acceso 22/05/2016]. URL: *https://docs.angularjs.org/guide*

[3] Ejemplos y ayuda AngularJS W3shools. [Último acceso 22/05/2016]. URL: *http://www.w3schools.com/angular/*

[4] Plugins AngularJS. [Último acceso 22/05/2016]. URL: *https://github.com/*

[5] Framework Bootstrap. [Último acceso 22/05/2016]. URL: *http://getbootstrap.com/*

[6] Ejemplos y ayuda Bootstrap W3shools. [Último acceso 22/05/2016]. URL: *http://www.w3schools.com/bootstrap/*

[7] Plugins Bootstrap para AngularJs. [Último acceso 22/05/2016]. URL: *https://angular-ui.github.io/bootstrap/*

[8] Iconos FontAwsome. [Último acceso 22/05/2016]. URL: *http://fontawesome.io/icons/*

[9] Editor Online Front-End Plunker. [Último acceso 22/05/2016]. URL: *https://plnkr.co/*

[10] Comunidad de resolución Y consulta de problemas Stackoverflow. [Último acceso 22/05/2016]. URL:*http://stackoverflow.com/*

[11] RESTful webApi. [Último acceso 22/05/2016]. URL: *http://www.asp.net/web-api/*

[12] Ejemplos y ayuda SQL. [Último acceso 22/05/2016]. URL: *http://www.w3schools.com/sql/*

[13] LINQ de C#. [Último acceso 22/05/2016]. URL: *https://msdn.microsoft.com/es-es/library/bb397933.aspx*

[14] Herramienta de pruebas JMeter. [Último acceso 22/05/2016]. URL: *http://jmeter.apache.org/*

[15] Definiciones específicas para el apéndice Wikipedia. [Último acceso 22/05/2016]. URL: *https://es.wikipedia.org/*

Glosario

AngularJs	Framework de JavaScript para la creación de aplicaciones web con la filosofía SPA. Su objetivo principal es aumentar las aplicaciones del lado Cliente con el patrón MVC reduciendo la carga de trabajo al lado Servidor.
Ajax	Acrónimo de *Asynchronous JavaScript And XML*, es una técnica de desarrollo web para crear aplicaciones interactivas.
Árbol	Estructura de datos de nodos conectados que imita la forma jerárquica de los árboles comunes. De esta forma se organizan con una relación de padres e hijos.
Array	Forma de almacenamiento continuo que contiene una serie de elementos del mismo tipo.
ASP.NET	Framework para el desarrollo de aplicaciones web creado por Microsoft.
Blob	Sistema de almacenamiento de objetos binarios de gran tamaño.
C++	Lenguaje de programación de los 80, cuya intención era extender el lenguaje C con mecanismos de manipulación de objetos.
Caché	Sistema de almacenamiento de acceso rápido que guarda temporalmente ciertos datos. En este proyecto se usa una caché en lado Servidor, para no acceder continuamente a la base de datos.
Cliente	Aplicación informática que consume un servicio remoto de otro ordenador (Servidor). En este proyecto, el cliente es el navegador web que muestra la interfaz, procesa datos y realiza peticiones al Servidor RESTful WebApi.
Content server	Sistema de gestión de contenidos.
CSS	Lenguaje para definir y crear estilos de un documento estructurado (HTML).
Directiva	Tag que determina el comportamiento de un elemento HTML. En este proyecto de usan directivas básicas de HTML (* *, *<button>*, *<a>*), directivas propias del framework de AngularJS (*ng-repeat,ng-click, ng-include*) y directivas creadas para fines determinados (*pagination-control*).
Enrutado	Sistema de navegación y organización identificado por estados, que determina los bloques y sus elementos de la vista de la aplicación.

Experiencia de Usuario	Conjunto de factores y elementos referentes a la interacción del usuario con un entorno, cuyo resultado es la percepción positiva o negativa de dicho producto, servicio o dispositivo. En este proyecto, una de las finalidades principales es mejorar la Experiencia de Usuario mediante la re modelación del software.
Expresión labda	Concepto matemático de los 80 aplicado a los lenguajes de programación. En este proyecto se usa en el lenguaje C# para realizar búsquedas de objetos en un *array* determinado.
Framework	Estructura conceptual y tecnológica basada en funcionalidades y módulos concretos para ayudar a desarrollar un proyecto software.
Front-end	Parte Cliente de una aplicación informática, es el conjunto de funcionalidades que interactúan con los usuarios.
GRID	Forma de representación de datos en tabla. Se compone de filas, columnas y celdas.
HTTP	Protocolo de información para la comunicación de elementos en la web.
Html	Lenguaje de marcado para la elaboración de páginas web.
Inyección SQL	Método de infiltración de código intruso para realizar operaciones sobre una base de datos.
Inyección de Dependencia	Patrón de diseño orientado a objetos en el que se suministran objetos de una clase en lugar de ser la propia clase quien cree el objeto. Es una forma de reutilización de componentes. En este proyecto se usa para la reutilización de componentes de módulos de Angular.
Javascript	Lenguaje de programación orientado a objetos, débilmente tipado y dinámico.
JSON	Formato de texto ligero para el intercambio de datos.
LINQ	Componente del lenguaje .NET para la consulta de datos.
Motor ECM	Organización y conjunto de componentes para la gestión del contenido empresarial.
Paginación	Numeración de páginas de un documento. En este proyecto se utiliza para dividir el listado de información en páginas.
PixelwareApi	Api de la empresa Pixelware que comprende las librerías necesarias para la interacción con la base de datos y el *Content Server*.

Plugins	Componentes Bootstrap para la representación de elementos. En este proyecto se utilizan multitud de componentes para determinadas necesidades, algunos de los más utilizados son *ui-tree, ui-view, ng-dialog, uib-datepicker-popup*.
Predicate	Expresión de programación para expresar condiciones expecíficas.
Responsive	Diseño web adaptable cuyo objetivo es adaptar las páginas web al dispositivo que las esté visualizando.
RESTful	Estilo de arquitectura software para el traspaso de elementos en la web.
Scope	Contexto de un *Controller* en Angular. Contiene una serie de atributos para manejar los datos entre la vista y el controlado.
Serialización	Codificación de un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión en red. En este proyecto se usa JSON para el traspaso de información entre Cliente y Servidor.
Servidor	Aplicación en ejecución capaz de atender las peticiones del Cliente y devolverle una respuesta en concordancia.
Single Page Application (SPA)	Aplicación web en una única página con el objetivo de dar al usuario una experiencia de navegación más fluida.
Sql	Lenguaje declarativo de acceso a bases de datos.
SpaguettiCode	Programa informático con código desordenado, complejo e incomprensible.
Tooltip	Herramienta de ayuda visual mostrada al pasar con el ratón sobre un elemento determinado.
URI	Identificador de recursos uniforme. En este proyecto se usa para identificar las funciones que ofrece el servidor RESTful webApi.
Vh	Unidad de medida para elementos HTML en el fichero de estilos que proporciona CSS3.

Anexos

A Script SQL

```
--CREACION TABLA DE REFERENCIA
IF NOT EXISTS (SELECT * FROM sysobjects WHERE id = OBJECT_ID(N'PWSAUXILIARES') AND
((OBJECTPROPERTY(id, N'IsUserTable') = 1) OR (OBJECTPROPERTY(id, N'IsView') = 1)))
BEGIN
CREATE TABLE PWSAUXILIARES(
	Referencia varchar(20) NOT NULL,
	Titulo varchar(50) NOT NULL,
	Descripcion varchar(100),
	Tipo bit,
PRIMARY KEY
(
	Referencia
)
)

--INSERCIÓN DE TABLAS
INSERT INTO PWSAUXILIARES (Referencia, Titulo, Descripcion, Tipo)
		SELECT name,name+'_TITULO',name+'_DESCRIPCION', 0
		from sys.tables
		where name like 'PWAS%'
	UNION
		SELECT name,name+'_TITULO',name+'_DESCRIPCION', 1
		from sys.tables
		where name like 'PWJS%'
END
GO
```

Para cada *Tabla Asociada* (PWAS…) o *Jerárquica* (PWJS…) se inserta un nuevo valore en la nueva Tabla Auxiliar *PWSAUXILIARES.* Como anteriormente no tenían título ni descripción, se fija uno temporal añadiendo _TITULO o _DESCRIPCION. En las posteriores creaciones de Tablas Asociadas o Jerárquicas, el usuario a fijará un título con la nueva interfaz para crear tablas.

B Función recursiva simulación de búsquedas en árboles

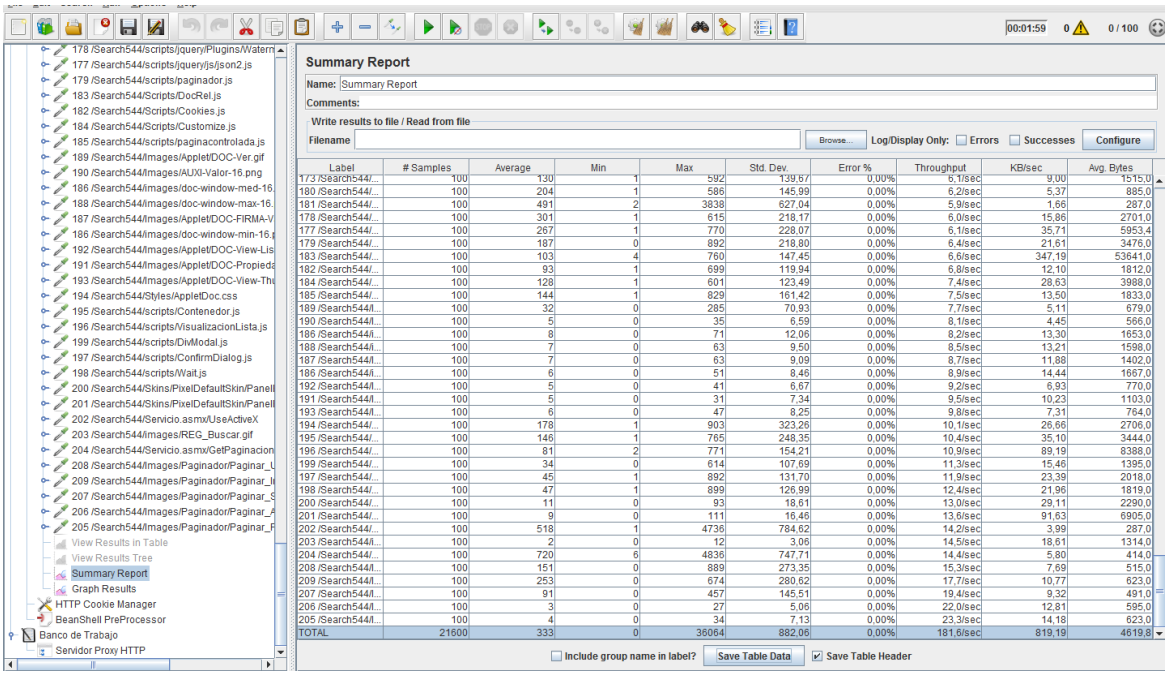
```
// Actualiza el árbol desplegando en el los nodos necesarios para mostrar el
//resultado de una búsqueda
// El árbol de $scope tiene la misma estructura de niveles que el devuelto por el
//Servidor "result" de id=0 a id=x
// result es un árbol de niveles desde el nodo root hasta el nodo que se ha buscado
//(sin hermanos).
var updateTreeFromSearchResult = function (nodes: Array<CommonTreeNode>, result:
Array<CommonTreeNode>): void {

   for (var i = 0; i < nodes.length; i++) {
      if (nodes[i].id == result[0].id) {
// CASO 1: se trata de un nodo padre, miramos si tenemos que expandirlo
         if (!result[0].collapsed && result[0].hasChilds) {
// CASO 1.1: padre expandido, seguimos la jerarquía llamando recursivamente
            if (nodes[i].hasChilds && !nodes[i].collapsed) {
               updateTreeFromSearchResult(nodes[i].nodes, result[0].nodes);
            }
//CASO 1.2: padre no expandido, lo expandimos y seguimos la jerarquía recursivamente
            else if (nodes[i].hasChilds && nodes[i].collapsed) {
//Guardo el nodo a expandir en una variable, (nodes[i])
               var expandedNode = nodes[i];
               var parent = parseInt(expandedNode.id);

//Traigo sus hijos del Servidor
               nodeService.getHierarchicalValuesTree(
                        $scope.selectedAuxiliarTable.reference, null,
                        $scope.searchClick,parent)
                  .then(function (resultNode: HierarchicalAuxiliarTableResult) {
                  expandedNode.nodes = resultNode.values;
                  expandedNode.collapsed = false;
                  expandedNode.hasChilds = true;

//Una vez obtenidos sus hijos, seguimos con la recursividad de la función
                  updateTreeFromSearchResult(expandedNode.nodes, result[0].nodes);
               });
            }
         }
// CASO 2: se trata de un nodo hijo, ya hemos llegado al resultado de la búsqueda
// o CASO 3: se trata de un nodo padre, no hay que expandirlo porque ya hemos
llegado al resultado de la búsqueda
            else if ((!nodes[i].hasChilds && !result[0].hasChilds) ||
                     (result[0].collapsed && result[0].hasChilds)) {
               $scope.selectedHierarchicalAuxiliarTableValue = nodes[i];
               $scope.selectedAuxiliarTableValue = nodes[i].value;
               goToItem($scope.selectedHierarchicalAuxiliarTableValue.id);
               break;
            }
      }
   }
}
```

C Summary Report Jmeter

A continuación se muestran las capturas realizadas sobre las pruebas de rendimiento que se han efectuado con *Apache JMeter*.

Se aprecian todas las peticiones y carga de recursos que son necesarias en cada caso, así como el tiempo y los valores *Throughput* y *KB/sec* necesarios para realizar la tabla de pruebas de rendimiento de la sección de pruebas.

Software Obsoleto Search



Software remodelado SolutionsSpa